C Language

Unit- I

Overview of programming languages: Generation and classification. Processes involved in program execution: Compilation, interpretation, loading, and linking. Flow charts and Algorithms. Basics of C Programming: Introduction to C programming language. Structure of a C program, C tokens, data types, variables, constants, operators, expression evaluation (precedence, associativity), type conversions in C.

Unit- II

Input and Output: Non-formatted and formatted input/output functions. Escape sequences and their usage in I/O. Control Statements: Selection statements if, if-else, nested if, switch, conditional operators. Iterative statements while, for, do-while. Special control statements goto, break, continue, return, exit. Arrays and Strings: One-dimensional arrays, Multidimensional arrays character arrays, ctype functions and string functions.

Unit -III

Functions: Function definition, declaration, and calling mechanisms, types of functions, Call-by-value, call-by-reference. Passing arrays to functions, recursion, inline functions. Scope and lifetime of variables, storage classes. Pointers: Introduction to pointers, address-of operator (&), pointer types. Pointers and arrays, pointers and strings, pointer to pointer, array of pointers. Dynamic memory allocation malloc, calloc, free.

Unit -IV

User-Defined Data Types: Structures and unions: Definition, initialization, accessing members.Arrays of structures, structures vs. unions. Enumeration types (enum). File handling: Introduction to file operations in C. File functions open, close, read, and writes. Working with text and binary files.

Textbook:

1. Pradip Dey, Manas Ghosh, Computer Fundamentals and Programming in C (2e)
2. Reema Thareja, "Programming in C", Oxford University Press, Second Edition, 2016.
3. Kernighan, B.W and Ritchie,D.M, "The C Programming language", Second Edition,

References:

1. Ivor Horton, Beginning C
2. Ashok Kamthane, Programming in C
3. Herbert Schildt, The Complete Reference C
4. Paul Deitel, Harvey Deitel, C How to Program
5. Byron S. Gottfried, Theory and Problems of Programming with C
6. Brian W. Kernighan, Dennis M. Ritchie, The C Programming Language
7. B. A. Forouzan, R. F. Gilberg, A Structured Programming Approach Using C

UNIT – I

UNIT – II

UNIT – I : Programming Fundamentals and Basics of C

Programming Language

A **programming language** is a formal language consisting of a set of symbols, rules, and instructions used to communicate with a computer system. It enables programmers to write programs that instruct a computer to perform specific tasks such as calculations, data processing, and decision making.

Programming languages act as an interface between humans and computers by converting human-readable instructions into machine-executable commands.

Generations of Programming Languages

The development of programming languages has progressed through several stages known as **generations**. Each generation represents a significant improvement in abstraction level, ease of programming, efficiency, and portability. Broadly, programming languages are classified into **five generations**, from machine-oriented languages to problem-oriented and intelligent languages.

1.First Generation Languages (1GL) – Machine Language

First Generation Languages (1GL), also known as **Machine Languages**, are the earliest form of programming languages. These languages consist of instructions written in **binary digits (0s and 1s)**, which are directly understood and executed by the computer's central processing unit (CPU) without the need for any translator.

Machine Language

Machine language instructions are specific to a particular computer architecture. Each instruction corresponds to a unique operation such as data movement, arithmetic computation, or control transfer.

Example

10101100 11001001

Characteristics of First Generation Languages

- Instructions are written in binary format (0 and 1)
- Directly executed by the hardware
- No compiler or interpreter is required
- Machine-dependent
- Very low-level language

Advantages of Machine Language

1. **High Execution Speed**
   Programs run very fast as they are directly executed by the CPU.

2. **Efficient Use of Memory**
   No additional memory is required for translators.
3. **Complete Hardware Control**
   Programmers have direct control over system resources.

Limitations of Machine Language

1. **Difficult to Learn and Use**
   Writing programs in binary form is extremely complex.
2. **Error-Prone**
   Even a single-bit error can cause program failure.
3. **Difficult Debugging and Maintenance**
   Identifying and correcting errors is very challenging.
4. **Machine Dependency**
   Programs written for one machine cannot run on another.
5. **Time-Consuming Development**
   Program development takes a long time.

Applications of Machine Language

- Early computer systems
- Microprocessor-level programming
- Hardware-specific operations

2. Second Generation Languages (2GL) – Assembly Language

Second Generation Languages (2GL) are known as **Assembly Languages**. These languages were developed to overcome the difficulties of machine language by using **symbolic instructions**, called **mnemonics**, instead of binary codes. Assembly language programs are translated into machine code using a software tool known as an **assembler**.

Assembly Language

Each assembly instruction corresponds to a single machine-level instruction. Although assembly language is easier to understand than machine language, it remains **machine-dependent**.

Example of Assembly Instructions:

MOV A, B
ADD A, C
SUB D, A

Assembler

An **assembler** converts assembly language instructions into equivalent machine code that can be executed by the computer.

Characteristics of Second Generation Languages

- Uses mnemonic codes instead of binary numbers

- Requires an assembler for translation
- One-to-one correspondence with machine instructions
- Machine-dependent
- Low-level programming language

Advantages of Assembly Language

1. **Improved Readability**
   Mnemonics make programs easier to read and understand.
2. **Better Debugging**
   Errors are easier to locate compared to machine language.
3. **Efficient Use of Hardware**
   Provides direct access to registers and memory.
4. **Faster Execution**
   Programs execute quickly as they are close to hardware.

Limitations of Assembly Language

1. **Machine Dependency**
   Programs written for one processor cannot be executed on another.
2. **Complexity**
   Requires detailed knowledge of hardware architecture.
3. **Time-Consuming Development**
   Writing large programs is difficult and lengthy.
4. **Poor Portability**
   Programs must be rewritten for different machines.

3. Third Generation Languages (3GL) – High-Level Languages

Third Generation Languages (3GL) are **high-level programming languages** designed to make program development easier, faster, and more reliable. These languages use **English-like syntax and mathematical notations**, allowing programmers to focus on problem-solving rather than hardware details.

3GL programs are **machine-independent** and require a **compiler or interpreter** to translate them into machine language.

Third Generation Languages

First and second generation languages were difficult to learn and were machine-dependent. To overcome these limitations, third generation languages were developed with:

- Improved readability
- Portability
- Reduced development time
- Better program maintenance

Examples of Third Generation Languages

- C
- C++
- Java
- Python
- FORTRAN
- Pascal

Characteristics of Third Generation Languages

- High-level and user-friendly
- English-like keywords and syntax
- Machine-independent
- Uses compilers or interpreters
- Supports structured programming
- Easier debugging and maintenance

Advantages of Third Generation Languages

1. **Ease of Learning and Use**
   Programs are easier to write, read, and understand.
2. **Portability**
   Same program can run on different machines with minimal changes.
3. **Faster Program Development**
   Reduces coding effort and development time.
4. **Better Maintenance**
   Errors are easier to locate and correct.
5. **Support for Large Applications**
   Suitable for business, scientific, and system software.

Limitations of Third Generation Languages

1. **Lower Execution Speed**
   Slower than machine and assembly languages.
2. **Less Hardware Control**
   Limited direct access to system resources.

Special Case: C Language

C is often referred to as a **middle-level language** because it combines:

- High-level features (structured programming, portability)
- Low-level features (pointers, memory access)

4. Fourth Generation Languages (4GL)

Fourth Generation Languages (4GL) are **very high-level, non-procedural programming languages** designed to simplify and accelerate software development. In these languages, the

programmer specifies **what result is required** rather than **how the task should be performed**.

4GLs are mainly used in **database management, data analysis, and business applications**.

Fourth Generation Languages

As software systems became larger and more complex, there was a need to:

- Reduce development time
- Minimize coding effort
- Enable non-programmers to interact with computers

4GLs address these needs by providing powerful built-in functionalities.

Examples of Fourth Generation Languages

- SQL (Structured Query Language)
- MATLAB
- SAS
- Oracle Forms

Characteristics of Fourth Generation Languages

- Very high-level abstraction
- Non-procedural or declarative in nature
- Minimal coding required
- Easy to learn and use
- Machine-independent

Advantages of Fourth Generation Languages

1. **Rapid Application Development**
   Applications can be developed quickly with fewer lines of code.
2. **User-Friendly**
   Suitable even for users with limited programming knowledge.
3. **High Productivity**
   Reduces programming effort and cost.
4. **Strong Database Support**
   Widely used in data manipulation and reporting.

Limitations of Fourth Generation Languages

1. **Limited Flexibility**
   Not suitable for system-level programming.
2. **Lower Performance**
   Execution may be slower compared to 3GLs.
3. **Dependency on Tools**
   Requires specialized software environments.

5 Fifth Generation Languages (5GL)

Fifth Generation Languages (5GL) are advanced programming languages designed primarily for **artificial intelligence (AI)** and **knowledge-based systems**. These languages focus on **problem-solving using logic, constraints, and rules**, rather than writing step-by-step procedures.

In 5GLs, the programmer specifies **what problem needs to be solved**, and the language system determines **how to solve it**.

Fifth Generation Languages

With the growth of AI, expert systems, and intelligent computing, traditional procedural languages became insufficient. Fifth Generation Languages were developed to:

- Support intelligent decision-making
- Reduce algorithmic complexity
- Enable machine reasoning

Examples of Fifth Generation Languages

- Prolog
- Mercury
- AI-based and logic-programming languages

Characteristics of Fifth Generation Languages

- Based on **logic programming** and **constraints**
- Declarative in nature
- Focus on problem description rather than algorithm design
- Suitable for artificial intelligence applications
- Machine-independent

Advantages of Fifth Generation Languages

1. **Supports Artificial Intelligence**
   Ideal for expert systems, robotics, and natural language processing.
2. **Reduced Coding Effort**
   Programmers define rules and constraints instead of detailed logic.
3. **High Level of Abstraction**
   Simplifies complex problem-solving tasks.

Limitations of Fifth Generation Languages

1. **Complex Implementation**
   Requires sophisticated runtime systems.
2. **Limited General-Purpose Use**
   Not suitable for conventional application development.
3. **Lower Performance**
   Execution speed may be slower than lower-level languages.

Feature	1GLMachine Language	2GLAssembly Language	3GLHigh-Level Language	4GLVery High-Level Language	5GLAI-Based Language
Level of Abstraction	Very Low	Low	High	Very High	Highest
Programming Style	Binary instructions	Mnemonics	Procedural / Structured	Non-procedural	Declarative / Logical
Language Dependency	Machine-dependent	Machine-dependent	Machine-independent	Machine-independent	Machine-independent
Ease of Use	Very Difficult	Difficult	Easy	Very Easy	Easy (conceptual)
Readability	Very Poor	Poor	Good	Very Good	Very High
Translator Required	No	Assembler	Compiler / Interpreter	Built-in tools	AI engine / Interpreter
Execution Speed	Very Fast	Fast	Moderate	Slower	Slowest
Development Time	Very High	High	Moderate	Low	Very Low
Debugging	Very Difficult	Difficult	Easy	Very Easy	Concept-based
Portability	None	Very Low	High	High	High
Hardware Control	Full	High	Limited	Very Limited	None
Typical Applications	Hardware control	Embedded systems	General applications	Business & databases	Artificial Intelligence
Examples	Binary code	Assembly	C, Java, Python	SQL, MATLAB	Prolog

Classification of Programming Languages

Programming languages can be classified in different ways based on their **programming paradigm**, **application area**, and **method of execution**. This classification helps in understanding the nature, purpose, and usage of various programming languages.

Classification of Programming Languages

1. ***Procedural Programming Languages***
2. ***Object-Oriented Programming Languages***
3. ***Functional Programming Languages***
4. ***Scripting Programming Languages***
5. ***Logic Programming Languages***
6. ***Compiled Programming Languages***

7. ***Interpreted Programming Language***

Procedural Programming Languages

Procedural programming languages are a class of programming languages in which a program is designed as a sequence of well-defined steps or procedures that operate on data. The main emphasis is on **how a problem is solved**, using a structured and systematic approach. Programs written in procedural languages are typically divided into functions or procedures, each responsible for performing a specific task, which helps in organizing code and improving readability.

In procedural programming, execution follows a **top-down approach**, where the main program controls the flow of execution by calling various procedures. These languages make extensive use of variables, loops, conditional statements, and function calls. Data and procedures are usually treated separately, and the same data may be accessed by multiple functions.

Procedural programming languages are widely used for system software, scientific applications, and general-purpose programming because of their efficiency and simplicity. They are especially suitable for problems that can be solved using clear, step-by-step logic.

Examples: C, FORTRAN, Pascal.

Object-Oriented Programming Languages

Object-Oriented Programming Languages are programming languages that organize programs around **objects**, which represent real-world entities. An object combines both **data (attributes)** and **methods (functions)** that operate on that data into a single unit. This approach helps in designing programs that are modular, reusable, and easier to maintain.

Object-oriented languages are based on key principles such as **encapsulation**, which hides internal details of an object; **inheritance**, which allows one class to acquire the properties of another; and **polymorphism**, which enables the same operation to behave differently in different contexts. These features improve code reusability, reduce complexity, and support large-scale software development.

In object-oriented programming, programs are structured as collections of interacting objects rather than sequences of instructions. This makes such languages especially suitable for developing complex applications, including enterprise software, graphical user interfaces, and web applications.

Examples: C++, Java.

Functional Programming Languages

Functional programming languages are a class of programming languages in which computation is treated as the evaluation of **mathematical functions**. These languages

emphasize the use of functions to perform operations and avoid changing program state or mutable data. Instead of using loops and variable assignments, functional programming relies heavily on **function calls and recursion**.

In functional programming, functions are treated as first-class entities, meaning they can be passed as arguments, returned from other functions, and stored in variables. This programming style leads to programs that are more predictable, easier to test, and simpler to reason about. Functional languages also support concepts such as immutability and higher-order functions.

Functional programming languages are commonly used in artificial intelligence, data processing, and academic research where complex computations and mathematical models are involved.

Examples: LISP, Haskell.

Scripting Programming Languages

Scripting programming languages are high-level languages primarily used for **automation, rapid application development, and task control**. These languages are generally **interpreted**, meaning the code is executed line by line without prior compilation. This makes scripting languages easy to write, test, and modify.

Scripting languages use simple and flexible syntax, allowing programmers to develop programs quickly with fewer lines of code. They are widely used to automate repetitive tasks, control other software applications, and develop dynamic web content. Because of their ease of use, scripting languages are popular among both beginners and professionals.

Scripting programming languages are commonly used in web development, system administration, and software testing.

Examples: Python, JavaScript.

Logic Programming Languages

Logic programming languages are a class of programming languages based on **formal logic and reasoning**. In these languages, programs are written using **facts, rules, and queries**, and the system uses logical inference to derive solutions. The programmer specifies **what conditions must be satisfied**, rather than writing step-by-step procedures to solve a problem.

Logic programming languages follow a **declarative programming approach**, where the focus is on defining relationships and constraints. The execution mechanism automatically determines how to apply rules to reach a solution. This makes logic programming particularly suitable for problems that involve reasoning, decision-making, and knowledge representation.

Logic programming languages are mainly used in **artificial intelligence, expert systems, and natural language processing**.

Example: Prolog.

Compiled Programming Languages

Compiled programming languages are languages in which the **entire source program is translated into machine code** before execution. This translation is performed by a software tool called a **compiler**. Once compiled, the program becomes an executable file that can be run multiple times without recompilation.

Because the complete program is converted into machine language in advance, compiled programs generally execute **faster and more efficiently** than interpreted programs. Errors are detected during the compilation stage, which helps in identifying syntax errors before program execution.

Compiled programming languages are widely used for system software, application development, and performance-critical programs.

Examples: C, C++.

Interpreted Programming Languages

Interpreted programming languages are languages in which the **source code is executed line by line** by an **interpreter**. Unlike compiled languages, the entire program is not converted into machine code before execution. Instead, each instruction is translated and executed at runtime.

This method of execution makes interpreted languages easier to debug, as errors are reported immediately at the line where they occur. Interpreted languages also allow greater flexibility and faster development, since programs can be run without a separate compilation step. However, because translation happens during execution, interpreted programs are generally slower than compiled programs.

Interpreted programming languages are widely used in web development, automation, and scripting applications.

Examples: Python, JavaScript

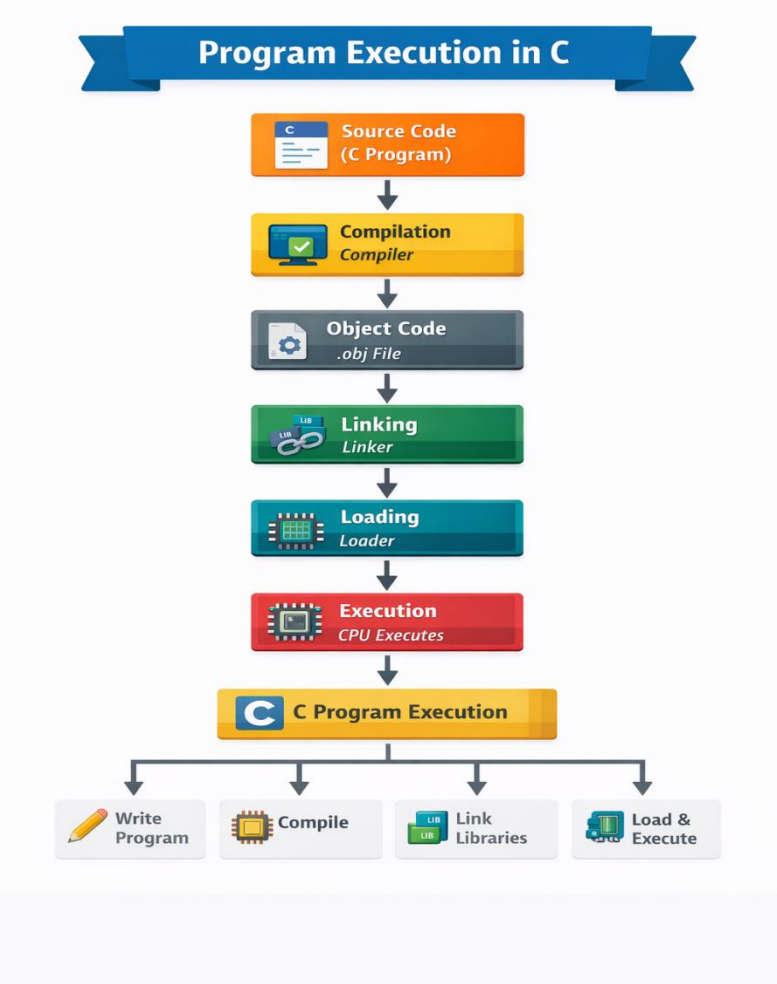
Steps Involved in Program Execution

1. Source Program Creation

The source program is written by the programmer using a high-level programming language such as C or Python. It contains human-readable instructions and is saved as a source file. This program cannot be executed directly by the computer.

2. Compilation (for Compiled Languages)

Compilation is the process of converting the entire source program into machine-level code using a compiler. During this step, syntax errors are detected and object code is generated. This step is applicable to compiled languages like C and C++.



3. Interpretation (for Interpreted Languages)

Interpretation is the process in which an interpreter translates and executes the program line by line. Errors are detected during execution, making debugging easier. This step is used in interpreted languages such as Python and JavaScript.

4. Linking

Linking combines the object code with required library files that contain predefined functions. The linker resolves external references and creates a complete executable file. This step ensures that all program components are correctly connected.

5. Loading

Loading is the process of placing the executable program into the main memory (RAM). The operating system allocates memory and prepares the program for execution. Necessary system resources are also assigned at this stage.

6. Execution

Execution begins when the CPU starts executing the program instructions. Control is transferred to the starting point of the program, usually the main() function. The program runs until it completes its task or terminates.

Flow Chart and Algorithm

Meaning of Flow Chart

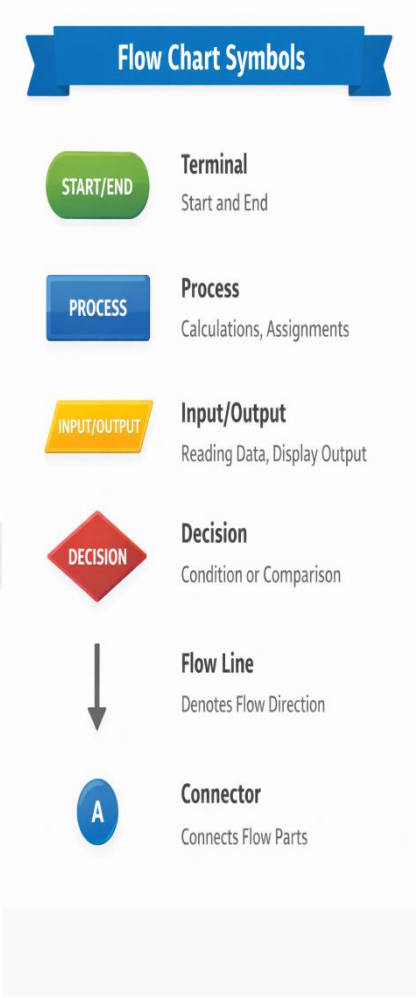
A **flow chart** is a graphical representation of an algorithm or process. It shows the step-by-step flow of execution using standard symbols connected by arrows. Flow charts help in understanding the logic of a program clearly before writing the actual code.

Flow charts are widely used in programming because they make problem solving easier, improve clarity, and help in detecting logical errors at an early stage.

Flow Chart Symbols

1. Terminal Symbol (Oval)

The terminal symbol is used to represent the **start** and **end** of a flow chart. It indicates where the process begins and where it stops.



2. Process Symbol (Rectangle)

The process symbol represents **calculations, assignments, or processing steps**. Any instruction such as arithmetic operations or data manipulation is shown using this symbol.

3. Input/Output Symbol (Parallelogram)

The input/output symbol is used for **reading input data or displaying output**. It represents operations like accepting values from the user or printing results.

4. Decision Symbol (Diamond)

The decision symbol represents a **condition or comparison**. It is used when a decision needs to be made, such as yes/no or true/false conditions, and it directs the flow accordingly.

5. Flow Line (Arrow)

Flow lines indicate the **direction of flow** from one step to another. They connect symbols and show the sequence of operations.

6. Connector Symbol (Circle)

The connector symbol is used to connect different parts of a flow chart. It is helpful when the flow chart is large and cannot fit on a single page.

Algorithm

Definition of Algorithm

An **algorithm** is a finite sequence of well-defined and unambiguous steps used to solve a specific problem or to perform a computation. Each step of an algorithm must be clear, precise, and executable, and the algorithm must produce a result after a finite number of steps.

Characteristics of an Algorithm

1. Input
An algorithm may accept **zero, one, or more inputs**, which are the data values required to solve the problem. These inputs must be **clearly specified** before the algorithm begins execution. Proper definition of input ensures that the algorithm works correctly for all valid cases and avoids ambiguity during processing.

2. Output
An algorithm must produce **at least one output**, which represents the result of the computation. The output should be **clearly related to the input** and must provide a meaningful solution to the problem. A well-designed algorithm always ensures that the expected output is obtained for every valid input.

3. Definiteness
Each step of an algorithm must be **precise, clear, and unambiguous**. There should be no confusion regarding what operation is to be performed at each step. Definiteness ensures that the algorithm behaves in the same manner every time it is executed with the same input.

4. Finiteness
An algorithm must **terminate after a finite number of steps**. It should not continue indefinitely. Finiteness guarantees that the algorithm reaches a conclusion and produces an output within a reasonable time, making it practically usable.

5. Effectiveness
All steps of the algorithm must be **simple, executable, and feasible** using available computational resources. Each operation should be basic enough to be carried out exactly and efficiently. This ensures that the algorithm can be implemented in a real programming environment without difficulty.

Advantages of an Algorithm

1. Algorithms help in understanding the logic of a problem before writing the actual program.

2. They provide a step-by-step solution, making problem-solving systematic and organized.
3. Algorithms are language-independent and can be implemented in any programming language.
4. They make program debugging and maintenance easier.
5. Algorithms improve clarity and accuracy in program development.

Disadvantages of an Algorithm

1. Writing algorithms for complex problems can be time-consuming.
2. Algorithms are not suitable for representing very complex logic visually.
3. They do not show program execution flow as clearly as flowcharts.
4. Converting large algorithms directly into code may be difficult.

Example: Algorithm to Find the Sum of Two Numbers

Step 1: Start
Step 2: Read two numbers A and B
Step 3: Calculate SUM = A + B
Step 4: Display the value of SUM
Step 5: Stop

Explanation

- **Input:** Two numbers A and B
- **Output:** Sum of the two numbers
- **Definiteness:** Each step is clearly defined
- **Finiteness:** Stops after a fixed number of steps
- **Effectiveness:** Uses simple and executable steps

Addition of Two Numbers

A) Algorithm Explanation

Algorithm to Add Two Numbers

Step 1: Start
Step 2: Read two numbers A and B
Step 3: Calculate SUM = A + B
Step 4: Display SUM
Step 5: Stop

Explanation

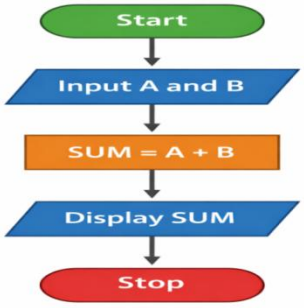
This algorithm provides a step-by-step logical solution to add two numbers. First, the program starts and accepts two input values. Then it performs the arithmetic operation of addition and stores the result in a variable. Finally, the result is displayed and the program terminates.

B) Flow Chart Explanation

Flow Chart for Adding Two Number

Explanation

The flow chart visually represents the same logic as the algorithm. Each step of the algorithm is shown using a standard flow chart symbol, connected by arrows to indicate the flow of execution. This graphical representation makes it easy to understand how the program executes step by step.



Introduction to C Programming Language

The **C programming language** is a general-purpose, procedural programming language developed in **1972 by Dennis M. Ritchie** at **Bell Laboratories**. It was originally designed for system programming, particularly for developing the UNIX operating system. Due to its efficiency, flexibility, and portability, C has become one of the most widely used programming languages.

C is often referred to as a **middle-level language** because it combines the features of both high-level languages and low-level languages. It supports structured programming concepts such as functions, loops, and conditional statements, while also providing low-level features like direct memory access through pointers.

The C programming language is **portable**, meaning programs written in C can be executed on different computer systems with minimal or no modification. It uses a rich set of operators and data types, which makes it suitable for developing system software, embedded systems, and application programs.

C is known for its **efficiency and speed**, as compiled C programs execute faster compared to interpreted languages. It also provides a standard library that includes a wide range of built-in functions for input/output operations, string handling, and mathematical computations.

Because of its simplicity and powerful features, C serves as the foundation for many modern programming languages such as C++, Java, and Python. It is widely used in operating systems, compilers, database systems, and real-time applications.

Features of C Programming Language

The **C programming language** is one of the most powerful and widely used programming languages. Its popularity is due to several important features that make it suitable for both system-level and application-level programming.

1. Simple and Efficient

C has a simple syntax with a limited number of keywords, which makes it easy to learn and use. Programs written in C are highly efficient and execute faster because they are compiled directly into machine code.

2. Structured Programming Language

C supports structured programming by allowing programs to be divided into functions and logical blocks. This improves program clarity, debugging, testing, and maintenance.

3. Middle-Level Language

C is often called a **middle-level language** because it supports both high-level programming features and low-level operations such as memory manipulation using pointers. This makes C suitable for system programming as well as application development.

4. Portability

C programs are portable, meaning they can be executed on different machines with little or no modification. Only the compiler needs to be changed for different platforms.

5. Rich Set of Operators

C provides a wide range of operators, including arithmetic, relational, logical, bitwise, and conditional operators. These operators allow efficient manipulation of data.

6. Extensive Standard Library

C has a powerful standard library that includes built-in functions for input/output operations, string handling, mathematical calculations, and file processing. This reduces the need to write programs from scratch.

7. Memory Management

C allows dynamic memory allocation using functions such as malloc(), calloc(), and free(). This gives programmers direct control over memory usage.

8. Fast Execution Speed

Since C is a compiled language, programs written in C execute faster compared to interpreted languages. This makes it suitable for performance-critical applications.

9. Modularity

C supports modular programming through user-defined functions. Large programs can be divided into smaller modules, improving readability and reusability.

10. Pointer Support

Pointers allow direct access to memory locations. This feature is powerful for system programming, array handling, and efficient data manipulation.

11. Recursion

C supports recursion, allowing functions to call themselves. This is useful for solving problems that can be broken into smaller sub-problems.

12. Widely Used and Supported

C is widely used in operating systems, embedded systems, compilers, databases, and real-time applications. Many modern languages like C++, Java, and Python are influenced by C.

Structure of a C Program

A **C program** follows a systematic structure that helps the compiler understand the program clearly and allows programmers to write organized, readable, and maintainable code. Each section of a C program has a specific purpose and role in execution.

General Structure of a C Program

```
/* Documentation Section */
#include <stdio.h>
#define MAX 100

/* Global Declarations */
```

```
int total;
void display();

/* main() Function */
int main() {
   display();
   return 0;
}

/* User-defined Functions */
void display() {
   printf("Hello C");
}
```

1. Documentation Section

The documentation section consists of **comments** that describe important information about the program such as the program name, purpose, author, and date. Comments are ignored by the compiler and do not affect the execution of the program. This section helps programmers understand the objective of the program easily.

Example:

/* Program to calculate sum of two numbers */

2. Link Section

The link section includes **header files** required by the program using the #include directive. Header files contain declarations of standard library functions and macros. Without including the necessary header files, library functions cannot be used in the program.

Example:

#include <stdio.h>

3. Definition Section

This section is used to define **symbolic constants** using the #define directive. These constants remain fixed throughout the program and improve code readability and maintainability.

Example:

#define MAX 100

4. Global Declaration Section

The global declaration section contains **global variables** and **function prototypes**. Global variables declared in this section can be accessed by all functions in the program. Function prototypes inform the compiler about the function name, return type, and parameters before their actual definition.

Example:

```
int total;
void display();
```

5. main() Function

The main() function is the **entry point of a C program**. Program execution always begins from the main() function. It contains local variable declarations, input/output statements, and function calls. The return 0; statement indicates successful program termination.

Example:

```
int main() {
   printf("Hello C Program");
   return 0;
}
```

6. User-Defined Functions

User-defined functions are functions written by the programmer to perform specific tasks. These functions help in **modular programming**, reduce code repetition, and improve program readability. They are usually defined after the main() function.

Example:

```
void display() {
   printf("Welcome to C Programming");
}
```

C Tokens

C tokens are the **smallest individual units** of a C program. They are the basic building blocks used to construct C statements and programs. The C compiler recognizes tokens as meaningful elements of the program.

Types of C Tokens

C tokens are broadly classified into the following categories:

- ***Keywords***
- ***Identifiers***
- ***Constants***
- ***String literals***
- ***Operators***
- ***Special symbols***

Keywords

Keywords are **reserved words** in the C programming language that have a **predefined meaning**. These words are part of the language syntax and are used to perform specific operations or define the structure of a C program. Keywords **cannot be used as identifiers**, such as variable names, function names, or array names.

C keywords are always written in **lowercase letters**, and their meaning cannot be changed by the programmer.

Examples of C Keywords

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

Categories of Keywords in C

- **Data Type Keywords:** int, float, char, double
- **Control Keywords:** if, else, switch, case
- **Loop Keywords:** for, while, do
- **Jump Keywords:** break, continue, goto, return
- **Storage Class Keywords:** auto, static, extern, register

Identifiers

Identifiers are the **names given to program elements** such as variables, functions, arrays, and user-defined data types in a C program. They are used to identify these elements uniquely during program execution.

Identifiers help programmers refer to data and functions in a readable and meaningful way.

Rules for Identifiers

1. An identifier must begin with a **letter (a–z or A–Z)** or an **underscore (_)**
2. It cannot begin with a digit
3. It can contain only letters, digits, and underscores
4. It must not be a **C keyword**
5. Identifiers are **case-sensitive** (total and Total are different)

Examples of Valid Identifiers

sum, total_marks, _count, value1

Constants

Constants are **fixed values** in a C program whose values **do not change during program execution**. Once a constant is defined, its value remains the same throughout the program.

Constants improve program readability and help avoid accidental modification of important values.

Types of Constants in C

1. Integer Constants

Integer constants are whole numbers without any fractional part. They can be positive or negative.

Examples:
10, -25, 0

2. Floating-Point Constants

Floating-point constants contain decimal points or exponential notation.

Examples:
3.14, 0.005, 2.5e3

3. Character Constants

Character constants consist of a **single character enclosed in single quotes**.

Examples:
'A', '9', '#'

4. String Constants (String Literals)

String constants are a sequence of characters enclosed in **double quotation marks**.

Examples:
"Hello", "C Programming"

5. Symbolic Constants

Symbolic constants are defined using the #define directive.

Example:

#define PI 3.14

String Literals

String literals are a sequence of characters enclosed within **double quotation marks (" ")**. They are treated as a **single token** by the C compiler and represent constant character data.

In C, a string literal is automatically terminated with a **null character ('\0')**, which indicates the end of the string.

Examples of String Literals

"Hello" "C Programming" "Welcome to C"

Characteristics of String Literals

- Enclosed in **double quotes**
- Stored as an array of characters
- End with a **null character ('\0')**
- Value **cannot be modified directly**
- Considered a **token**, but **not a data type**

Operators

Operators are special symbols used in a C program to perform operations on variables and constants. They help in manipulating data and evaluating expressions. Based on their functionality, operators in C are classified into different types.

Types of Operators in C

- ✓ **Arithmetic Operators**
  Used for mathematical calculations.
  Example: c = a + b;
- ✓ **Relational Operators**
  Used to compare two values.
  Example: a > b
- ✓ **Logical Operators**
  Used to combine logical conditions.
  Example: (a > b) && (b > c)
- ✓ **Assignment Operators**
  Used to assign values to variables.
  Example: x += 10;
- ✓ **Increment and Decrement Operators**
  Used to increase or decrease a value by one.
  Example: i++;
- ✓ **Conditional (Ternary) Operator**
  Used to choose between two values based on a condition.
  Example: max = (a > b) ? a : b;

- ✓ **Bitwise Operators**
  Used for bit-level operations.
  Example: a & b;

Special Symbols

Special symbols in C are characters that have **specific meanings** and are used to structure programs, separate statements, and define program blocks. They help the compiler understand the syntax and flow of a C program.

Special Symbols in C

;
{ }
( )
[ ]
,
:
#

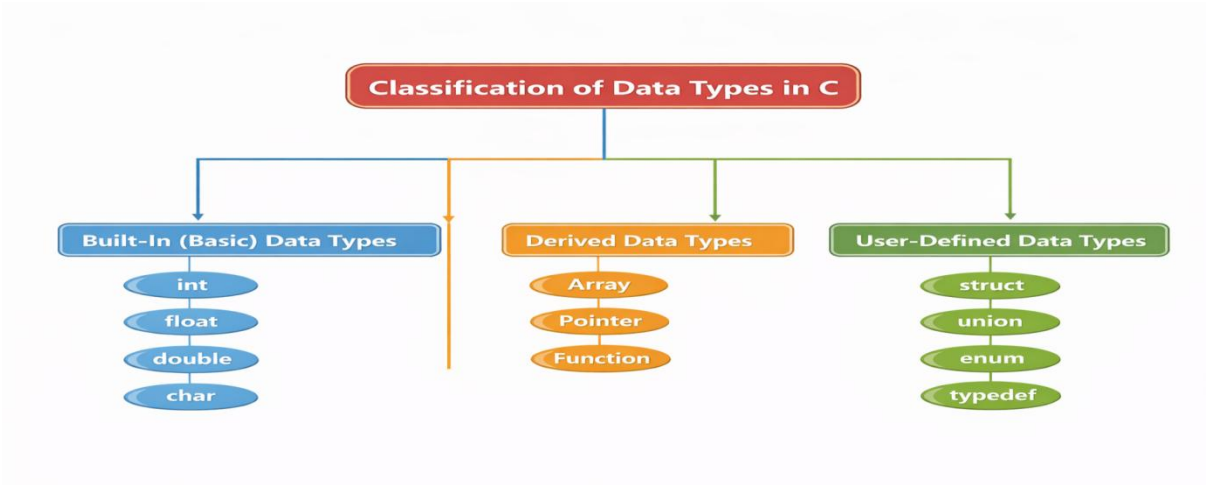
Data Types in C

Data types in C specify the **type of data** that a variable can store. They help the compiler understand how much memory to allocate for a variable and how the data should be processed.

Types of Data Types in C

Data types in C are classified into the following categories:

1. ***Basic (Primary) Data Types***
2. ***Derived Data Types***
3. ***User-Defined Data Types***



1. Basic (Primary) Data Types in C

Basic (Primary) data types are the fundamental data types provided by the C programming language. They are used to store **simple and single values** and form the foundation for all other data types. These data types define the type of data a variable can hold, the amount of memory allocated, and the range of values that can be stored.

int

The int data type is used to store **integer values**, which are whole numbers without any fractional part. It can store both positive and negative numbers as well as zero. The size of an int typically occupies **2 or 4 bytes**, depending on the system architecture. It is commonly used for counting, indexing, and control variables in loops.

Example:

```
int count = 10;
```

float

The float data type is used to store **decimal or floating-point values** with single precision. It generally occupies **4 bytes of memory** and provides up to **6 decimal digits of precision**. Float is suitable for applications where approximate values are acceptable, such as measurements and scientific calculations.

Example:

```
float price = 45.75;
```

double

The double data type is used to store **double-precision floating-point values**. It occupies **8 bytes of memory** and provides higher precision compared to float, typically up to **15 decimal digits**. Double is preferred in applications that require more accuracy, such as financial and scientific computations.

Example:

```
double distance = 12345.6789;
```

char

The char data type is used to store a **single character**, such as a letter, digit, or symbol. It occupies **1 byte of memory** and stores characters using **ASCII values**. The char data type is widely used in character handling and string manipulation.

Example:

char grade = 'A';

Void Data Type

The void data type represents **no value**. It is mainly used for functions that do not return a value.

Example:

void display();

Derived Data Types in C

Derived data types are data types that are formed by using **basic (primary) data types**. They allow programmers to store, manage, and process data in a more organized and efficient way. Derived data types help in handling complex data structures and improving program modularity.

Array

An **array** is a collection of **similar data elements** stored in **contiguous memory locations**. All elements of an array must be of the same data type. Arrays are useful when multiple values of the same type need to be stored and processed using a single variable name.

Example:

int marks[5] = {70, 80, 90, 85, 75};

Pointer

A **pointer** is a variable that stores the **memory address of another variable**. Pointers provide direct access to memory and are used for dynamic memory allocation, array manipulation, and efficient program execution.

Example:

int a = 10;
int *p = &a;

Function

A **function** is a group of statements that performs a **specific task**. Functions help divide a large program into smaller, manageable modules, making the program easier to understand, debug, and maintain.

Example:

```
int add(int x, int y)
 {
   return x + y;
}
```

3. User-Defined Data Types

User-defined data types allow programmers to **create their own data types** according to the requirements of a program. They help in organizing complex data, improving code readability, and making programs easier to manage and maintain.

struct

A **structure (struct)** is a user-defined data type that groups **different data types** under a single name. Each member of a structure has its **own separate memory location**, allowing all members to be used simultaneously.

Example:

```
struct student
{
   int id;
   char name[20];
   float marks;
};
```

union

A **union** is similar to a structure, but **all members share the same memory location**. Only one member of a union can store a value at any given time, which helps in saving memory.

Example:

```
union data
{
   int i;
   float f;
   char c;
};
```

enum

An **enumeration (enum)** is a user-defined data type that assigns **names to integer constants**. It improves program readability and reduces the use of numeric values directly in the code.

Example:

enum days {MON, TUE, WED, THU, FRI};

typedef

The typedef keyword is used to **create an alias (alternative name)** for an existing data type. It simplifies complex data type declarations and improves code clarity.

Example:

typedef int number;
number a = 10;

Variables in C

A **variable** is a named memory location used to store a value that can **change during program execution**. Each variable has a **data type**, which determines the type of value it can store and the amount of memory allocated to it. Variables help programmers store, process, and manipulate data efficiently.

Rules for Naming Variables

- ✓ A variable name must begin with a **letter (a–z or A–Z)** or an **underscore (_)**.
- ✓ A variable name **cannot begin with a digit**.
- ✓ Only **letters, digits, and underscores** are allowed in variable names.
- ✓ **Special characters** such as @, #, $, % are not allowed.
- ✓ Variable names **must not be C keywords**.
- ✓ Variable names in C are **case-sensitive** (total and Total are different).
- ✓ Variable names should be **meaningful and descriptive**.
- ✓ The length of a variable name should be **within compiler limits** (commonly 31 characters are significant).

Declaration of Variables

int count;
float salary;

Initialization of Variables

int a = 10; float rate = 5.5;

Types of Variables

- **Local Variables** – declared inside a function
- **Global Variables** – declared outside all functions

Constants in C

A **constant** is a value that **does not change during program execution**. Constants are used to represent fixed values in a program and help prevent accidental modification of important data.

Types of Constants

- **Integer Constants:** 10, -25
- **Floating-Point Constants:** 3.14, 0.5
- **Character Constants:** 'A', '9'
- **String Constants:** "Hello"
- **Symbolic Constants:** defined using #define

Example:

#define PI 3.14

Difference Between Variable and Constant

Variable	Constant
A variable is a named memory location used to store data.	A constant is a fixed value that does not change during program execution.
The value of a variable can be modified.	The value of a constant cannot be modified.
Declared using a data type.	Defined using fixed values or #define.
Requires memory allocation that can change during execution.	Stored as a fixed value in memory.
Used for storing changing data.	Used for storing permanent or fixed data.
Example: int x = 10;	Example: #define PI 3.14

Operators

An **operator** is a **symbol** that performs a specific operation on one or more operands (variables or constants). Operators are used to manipulate data, perform calculations, compare values, and control program flow in a C program.

Different Operators in C Language

1. ***Arithmetic Operators***
2. ***Relational Operators***

3. ***Logical Operators***
4. ***Assignment Operators***
5. ***Increment and Decrement Operators***
6. ***Conditional (Ternary) Operator***
7. ***Bitwise Operators***
8. ***Special Operators***

1. Arithmetic Operators

Arithmetic operators are used to perform basic mathematical calculations on variables and constants in a C program. These operators work on numeric data types such as int, float, and double.

Operator	*Operation*
+	*Addition*
-	*Subtraction*
*	*Multiplication*
/	*Division*
%	*Modulus (remainder)*

Examples

- int a = 10, b = 3;
- int sum = a + b; // Addition
- int diff = a - b; // Subtraction
- int prod = a * b; // Multiplication
- int quo = a / b; // Division
- int rem = a % b; // Modulus

Relational Operators

Relational operators are used to **compare two values or expressions**. The result of a relational operation is either **true (1)** or **false (0)**. These operators are commonly used in decision-making statements such as if, while, and for.

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

Examples

int a = 10, b = 20;

```
a < b;    // true
a > b;    // false
a == b;   // false
a != b;   // true
```

Logical Operators

Logical operators are used to **combine two or more conditions** or to **reverse a condition**. They are mainly used in decision-making and looping statements. The result of a logical operation is either **true (1)** or **false (0)**.

List of Logical Operators

1. && – Logical AND
2. || – Logical OR
3. ! – Logical NOT

Examples

```
int a = 10, b = 5, c = 20;

(a > b) && (a < c);   // true
(a > b) || (a > c);   // true
!(a > b);             // false
```

Assignment Operators

Assignment operators are used to **assign values to variables**. They can also perform an operation and assign the result to the same variable.

Operator	*Meaning*
=	*Assign*
+=	*Add and assign*
-=	*Subtract and assign*
*=	*Multiply and assign*
/=	*Divide and assign*
%=	*Modulus and assign*

Examples

```
int a = 10;

a += 5;   // a = a + 5
a -= 3;   // a = a - 3
a *= 2;   // a = a * 2
```

```
a /= 4;   // a = a / 4
a %= 3;   // a = a % 3
```

Increment and Decrement Operators

Increment and decrement operators are used to **increase or decrease the value of a variable by one**. These operators are commonly used in loops and counting operations.

List of Increment and Decrement Operators

Operator	Meaning
++	Increment
--	Decrement

Types

1. **Pre-Increment (++a)**
   Increases the value first, then uses it in the expression.
2. **Post-Increment (a++)**
   Uses the value first, then increases it.
3. **Pre-Decrement (--a)**
   Decreases the value first, then uses it.
4. **Post-Decrement (a--)**
   Uses the value first, then decreases it.

Examples
int a = 5;

```
++a;   // a becomes 6
a++;   // a becomes 7
--a;   // a becomes 6
a--;   // a becomes 5
```

Conditional Operator

The **conditional operator** is also known as the **ternary operator**. It is used to **evaluate a condition and return one of two values** based on whether the condition is true or false. It is the **only operator in C that takes three operands**.

Syntax

```
condition ? expression1 : expression2;
```

- If the condition is **true**, expression1 is executed
- If the condition is **false**, expression2 is executed

Example

```
int a = 10, b = 20;
int max;

max = (a > b) ? a : b;
```

Bitwise Operators

Bitwise operators are used to perform operations **directly on individual bits** of integer data. These operators work at the binary level and are mainly used in system programming, embedded systems, and performance-critical applications.

List of Bitwise Operators

1. & – Bitwise AND
2. | – Bitwise OR
3. ^ – Bitwise XOR
4. ~ – Bitwise NOT
5. << – Left Shift
6. >> – Right Shift

Examples

```
int a = 5, b = 3;

a & b;   // Bitwise AND
a | b;   // Bitwise OR
a ^ b;   // Bitwise XOR
~a;      // Bitwise NOT
a << 1;  // Left shift
a >> 1;  // Right shift
```

A	*B*	*A & B*	*A | B*	*A ^ B*
0	*0*	*0*	*0*	*0*
0	*1*	*0*	*1*	*1*
1	*0*	*0*	*1*	*1*
1	*1*	*1*	*1*	*0*

```
#include <stdio.h>
int main()
{
int a = 8;

printf("Original value of a = %d\n", a);

printf("Left Shift (a << 1) = %d\n", a << 1);
```

```
printf("Right Shift (a >> 1) = %d\n", a >> 1);

return 0;
}
```

Output

```
Original value of a = 8
Left Shift (a << 1) = 16
Right Shift (a >> 1) = 4
```

Special Operators

Special operators are operators in C that perform specific tasks other than arithmetic or logical operations. They are used for memory handling, size calculation, conditional evaluation, and accessing data.

List of Special Operators

1. **sizeof Operator**
2. **Comma Operator (,)**
3. **Conditional (Ternary) Operator (?:)**
4. **Address-of Operator (&)**
5. **Pointer (Indirection) Operator (*)**
6. **Member Access Operators (. and ->)**

- ✓ **sizeof** – Returns the size (in bytes) of a data type or variable
  Example: sizeof(int)
- ✓ **Comma (,)** – Allows multiple expressions in a single statement
  Example: a = (b = 5, b + 2);
- ✓ **Conditional (?:)** – Selects one of two values based on a condition
  Example: (a > b) ? a : b;
- ✓ **Address-of (&)** – Returns the memory address of a variable
  Example: &x
- ✓ **Pointer (*)** – Used to access the value stored at an address
  Example: *ptr
- ✓ **Member Access Operators**
- ✓ . → Access structure members
- ✓ -> → Access structure members using pointers

Expression evaluation is the process of computing the value of an expression by applying operators to operands according to **operator precedence** and **associativity** rules.

An **expression** is a combination of variables, constants, and operators.

Expression Evaluation Depends on Precedence and Associativity

In C, **expression evaluation** is the process of determining the final value of an expression. When an expression contains **multiple operators**, the order in which these operators are evaluated is **not random**. It depends on two important rules: **operator precedence** and **operator associativity**.

Operator precedence decides **which operator is evaluated first**. Operators with higher precedence are evaluated before operators with lower precedence. For example, in the expression 10 + 5 * 2, the multiplication operator (*) has higher precedence than addition (+), so 5 * 2 is evaluated first.

Operator associativity decides the **direction of evaluation** when two or more operators of the **same precedence** appear in an expression. Most arithmetic operators follow **left-to-right associativity**, while assignment operators follow **right-to-left associativity**. For example, in a = b = 5, the value 5 is first assigned to b and then to a.

Therefore, expression evaluation in C strictly follows the rules of **precedence first**, and **associativity next** to produce the correct result.

Operators	***Description***	***Associativity***
()	*Parentheses*	*Left to Right*
[]	*Array subscript*	*Left to Right*
->	*Structure pointer*	*Left to Right*
.	*Structure member*	*Left to Right*
++ --	*Increment, Decrement (post)*	*Left to Right*
++ --	*Increment, Decrement (pre)*	*Right to Left*
+ -	*Unary plus, Unary minus*	*Right to Left*
! ~	*Logical NOT, Bitwise NOT*	*Right to Left*
*	*Pointer dereference*	*Right to Left*
&	*Address-of*	*Right to Left*
sizeof	*Size operator*	*Right to Left*
** / %*	*Multiplication, Division, Modulus*	*Left to Right*
+ -	*Addition, Subtraction*	*Left to Right*
<< >>	*Left shift, Right shift*	*Left to Right*
< <= > >=	*Relational operators*	*Left to Right*
== !=	*Equality operators*	*Left to Right*
&	*Bitwise AND*	*Left to Right*
^	*Bitwise XOR*	*Left to Right*

`	`	*Bitwise OR*
&&	*Logical AND*	*Left to Right*
`		`
?:	*Conditional (ternary)*	*Right to Left*
*= += -= *= /= %=*	*Assignment operators*	*Right to Left*
,	*Comma operator*	*Left to Right*

Type Conversion

Type conversion in C is the process of converting a value from **one data type to another**. It is necessary when expressions involve different data types so that operations can be performed correctly.

Types of Type Conversion

1. **Implicit Type Conversion (Automatic Type Conversion)**
2. **Explicit Type Conversion (Type Casting)**

Implicit Type Conversion

Implicit type conversion, also known as **automatic type conversion**, is the process in which the **compiler automatically converts one data type into another** without any programmer intervention. This conversion usually happens when different data types are used together in an expression.

The compiler generally converts a **lower data type to a higher data type** to avoid loss of data.

```
int a = 10;
float b;

b = a;   // int is automatically converted to float
```

Explicit Type Conversion

Explicit type conversion, also known as **type casting**, is the process of **manually converting one data type into another** by the programmer. It is used when the programmer wants to **control the type of conversion** and the result of an expression.

```
int a = 5;
int b = 2;
float result;

result = (float)a / b;   // result = 2.5
```

Implicit Type Conversion	***Explicit Type Conversion***
Performed automatically by the compiler	*Performed manually by the programmer*
No cast operator is used	*Cast operator is used*
Converts lower data type to higher data type	*Can convert higher to lower data type*
Safer and avoids data loss	*May cause data loss*
Happens during expression evaluation	*Done intentionally by programmer*
Example: float x = 10;	*Example: int x = (int)3.5;*

Unit- II

Input and Output Functions in C

Input and Output (I/O) functions in C are used to **accept data from the user (input)** and **display results to the user (output)**. These functions enable interaction between the program and external devices such as the **keyboard (input device)** and **monitor (output device)**. In C, all standard I/O operations are provided through the **standard library header file** `<stdio.h>`.

I/O functions in C convert data between the **internal binary form used by the computer** and the **human-readable form** required for input and display. Based on the level of control over formatting, C classifies I/O functions into **non-formatted** and **formatted** input/output functions.

Classification of Input and Output Functions in C

1. ***Non-Formatted Input/Output Functions***
   - Do not use format specifiers
   - Handle simple character or string input/output
   - Examples: `getchar()`, `putchar()`, `gets()`, `puts()`
2. ***Formatted Input/Output Functions***
   - Use format specifiers to control data type and format
   - Provide better control over input and output
   - Examples: `scanf()`, `printf()`

1. *Classification of Non-Formatted I/O Functions*:

A. Character Input/Output Functions

- ***getchar()***
- ***putchar()***

B. String Input/Output Functions

- ***gets()***
- ***puts()***

A. Character Input/Output Functions

`getchar()`

`getchar()` is a **non-formatted input function** in the C programming language used to **read a single character from the standard input device (keyboard)**. It reads exactly one character at a time, including whitespace characters such as space, tab, and newline. The function waits for the user to press the **Enter key**, after which the entered character is returned. The value returned by `getchar()` is typically stored in a variable of type `char`. Since it does not use format specifiers, `getchar()` is simple to use and is commonly

employed in programs that require character-by-character input, such as menu-driven applications or basic text processing tasks.

Syntax

char ch;
ch = getchar();

Example:

```
#include <stdio.h>

int main()
{
   char ch;
   printf("Enter a character: ");
   ch = getchar();
   printf("You entered: %c", ch);
   return 0;
}
```

In this example, `getchar()` reads a single character entered by the user and stores it in the variable `ch`. The entered character is then displayed using `printf()`.

putchar()

`putchar()` is a **non-formatted output function** in the C programming language used to **display a single character on the standard output device (screen)**. It prints exactly one character at a time, which may be a character constant, a variable of type `char`, or the result of a character expression. The function is simple and efficient, and it is often used in programs that require character-by-character output, such as displaying characters inside loops or outputting processed input. Since `putchar()` does not use format specifiers, it provides a straightforward way to output individual characters.

Example:

```
#include <stdio.h>

int main() {
   char ch = 'A';
   putchar(ch);
   return 0;
```

```
}
```

In this example, `putchar()` prints the character stored in the variable `ch` to the screen.

B. String Input / Output Functions in C

String input/output functions in C are **non-formatted I/O functions** used to **read and display a sequence of characters (strings)**.

gets()

`gets()` is a **non-formatted input function** in the C programming language used to **read a string from the standard input device (keyboard)**. It reads characters continuously until the user presses the **Enter key**, and then stores the entire line of input in a character array. After reading the input, `gets()` automatically appends a **null character (`'\0'`)** at the end of the string to mark its termination. This function allows the input of strings containing **spaces**, making it useful for reading full sentences or names.

However, `gets()` does **not perform any boundary checking** on the input size. If the user enters more characters than the array can hold, it may cause **buffer overflow**, leading to unexpected behavior. For this reason, `gets()` is considered unsafe and has been removed from modern C standards, but it is often included in syllabi for conceptual understanding.

Example:

```
#include <stdio.h>

int main() {
    char name[30];
    printf("Enter your name: ");
    gets(name);
    printf("Your name is: ");
    puts(name);
    return 0;
}
```

In this example, `gets()` reads a complete line of text (including spaces) and stores it in the array `name`, which is then displayed using `puts()`.

puts()

`puts()` is a **non-formatted output function** in the C programming language used to **display a string on the standard output device (screen)**. It prints the characters of a string one by one until it encounters the **null character (`'\0'`)**, which marks the end of the string. After

displaying the string, `puts()` automatically moves the cursor to the **next line** by appending a newline character. This makes it convenient for printing text messages without manually adding `\n`.

The `puts()` function is simple to use and is mainly employed for **basic string output operations**. However, it does not provide formatting control and can print only one string at a time.

Example:

```
#include <stdio.h>

int main() {
   char msg[] = "Welcome to C Programming";
   puts(msg);
   return 0;
}
```

In this example, `puts()` displays the string stored in `msg` on the screen and automatically moves the cursor to the next line.

2. Formatted Input / Output Functions

Formatted input/output functions in C are used to **read and display data in a specified format**. These functions provide **greater control over the appearance and interpretation of data** by using **format specifiers**. They are widely used in C programs for handling numeric and textual data efficiently.

Formatted I/O functions are defined in the standard header file:

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

Types of Formatted I/O Functions

1. ***Formatted Input Function – `scanf()`***
2. ***Formatted Output Function – `printf()`***

scanf()

`scanf()` is a **formatted input function** in the C programming language used to **read data from the standard input device (keyboard)**. It allows the programmer to accept different types of data such as integers, floating-point numbers, characters, and strings by using **format specifiers**. The function reads input according to the specified format and stores the values in the memory locations of the given variables. Therefore, the **address-of operator**

(&) is used before variable names (except for strings) to provide their memory addresses. If the input does not match the format specifiers, `scanf()` stops reading further input, which may lead to incorrect results. This function is commonly used when precise control over input data types is required.

Syntax

```
scanf("format_specifier", &variable1, &variable2, ...);
```

Example:

```
#include <stdio.h>
int main()
{
   int age;
   float marks;

   printf("Enter age and marks: ");
   scanf("%d %f", &age, &marks);

   printf("Age = %d\n", age);
   printf("Marks = %.2f\n", marks);

   return 0;
}
```

In this example, `scanf()` reads an integer value for `age` and a floating-point value for `marks` from the keyboard and stores them in the respective variables based on the given format specifiers.

Common Format Specifiers

Format Specifier	***Data Type***
%d	*Integer*
%f	*Float*
%lf	*Double*
%c	*Character*
%s	*String*
%u	*Unsigned integer*

printf()

`printf()` is a **formatted output function** in the C programming language used to **display data on the output screen** in a specified format. It allows the programmer to print different types of data such as integers, floating-point numbers, characters, and strings by using

format specifiers within a format string. The function converts the internal binary representation of data into a human-readable form and displays it according to the given format. `printf()` also supports **escape sequences** (like `\n` for a new line and `\t` for a tab) and **width and precision control**, which help in presenting output neatly and clearly. It is one of the most commonly used functions in C for producing well-formatted and user-friendly output.

Syntax

```
printf("format_string", value1, value2, ...);
```

Example:

```
#include <stdio.h>

int main() {
    int roll = 101;
    float marks = 87.56;

    printf("Roll Number: %d\n", roll);
    printf("Marks: %.2f", marks);

    return 0;
}
```

In this example, `printf()` displays an integer value using `%d` and a floating-point value using `%.2f`, which prints the marks up to two decimal places, demonstrating formatted output in C.

Common Format Specifiers

Specifier	*Description*
%d	*Integer*
%f	*Floating-point number*
%c	*Character*
%s	*String*
%lf	*Double*

Escape Sequences in C and Their Usage in Input/Output

In C programming, **escape sequences** are special character combinations used within character constants and strings to represent **non-printable characters** or to **control the format of input and output**. Escape sequences begin with a **backslash (\)** followed by a specific character. They are mainly used with **formatted output functions like `printf()`** and also affect input/output display behavior.

Meaning of Escape Sequences

Escape sequences enable programmers to:

- Move the cursor to a new line or tab position
- Insert special characters in output
- Improve readability and formatting of displayed text
- Control how output appears on the screen

Common Escape Sequences in C

Escape Sequence	*Meaning*	*Usage in I/O*
\n	*New line*	*Moves cursor to next line*
\t	*Horizontal tab*	*Adds tab space*
\b	*Backspace*	*Moves cursor one position left*
\r	*Carriage return*	*Moves cursor to beginning of line*
\f	*Form feed*	*Page break*
**	*Backslash*	*Prints \ character*
\'	*Single quote*	*Prints '*
\"	*Double quote*	*Prints "*
\a	*Alert (beep)*	*Produces beep sound*
\0	*Null character*	*String termination*

1. New Line Escape Sequence (\n)

The escape sequence `\n` is used to **move the cursor to the beginning of the next line**. It is commonly used in output statements to display text on multiple lines and improve readability of the output.

Example:

```
printf("Hello\nWorld");
```

Output:

```
Hello
World
```

2. Horizontal Tab Escape Sequence (\t)

The escape sequence `\t` inserts a **horizontal tab space** in the output. It is useful for aligning text in columns, such as tables and formatted reports.

Example:

```
printf("Name\tMarks");
```

Output:

```
Name    Marks
```

3. Backspace Escape Sequence (\b)

The escape sequence `\b` moves the cursor **one position backward**, effectively deleting the previous character from the output. It is rarely used in modern programs but helps demonstrate cursor control.

Example:

```
printf("ABC\bD");
```

Output:

```
ABD
```

4. Carriage Return Escape Sequence (\r)

The escape sequence `\r` moves the cursor to the **beginning of the current line** without moving to a new line. Any text printed after `\r` overwrites the existing content.

Example:

```
printf("Hello\rWorld");
```

Output:

```
World
```

5. Form Feed Escape Sequence (\f)

The escape sequence `\f` is used to indicate a **page break**. In modern systems, its effect may not be visible, but it is conceptually used in printers and formatted documents.

Example:

```
printf("Page1\fPage2");
```

6. Backslash Escape Sequence (\\)

The escape sequence `\\` is used to **print a backslash character (\)** in the output, since a single backslash is treated as the beginning of an escape sequence.

Example:

```
printf("C:\\Program Files\\");
```

Output:

```
C:\Program Files\
```

7. Double Quote Escape Sequence (\")

The escape sequence `\"` is used to **print double quotation marks** within a string. Without this escape sequence, the compiler would treat the quote as the end of the string.

Example:
printf("He said \"Welcome to C\"");

Output:
He said "Welcome to C"

8. Single Quote Escape Sequence (\')

The escape sequence `\'` is used to **print a single quotation mark**, especially within character constants or strings.

Example:

```
printf("It\'s a C program");
```

Output:

```
It's a C program
```

9. Alert (Beep) Escape Sequence (\a)

The escape sequence `\a` generates an **alert or beep sound** when the program is executed. Its effect depends on the system.

Example:

```
printf("\aWarning!");
```

10. Null Character (\0)

The escape sequence `\0` represents the **null character**, which marks the **end of a string in C**. It is automatically added to strings and is not directly printed.

Example:

```
char name[] = "C"; pri
```

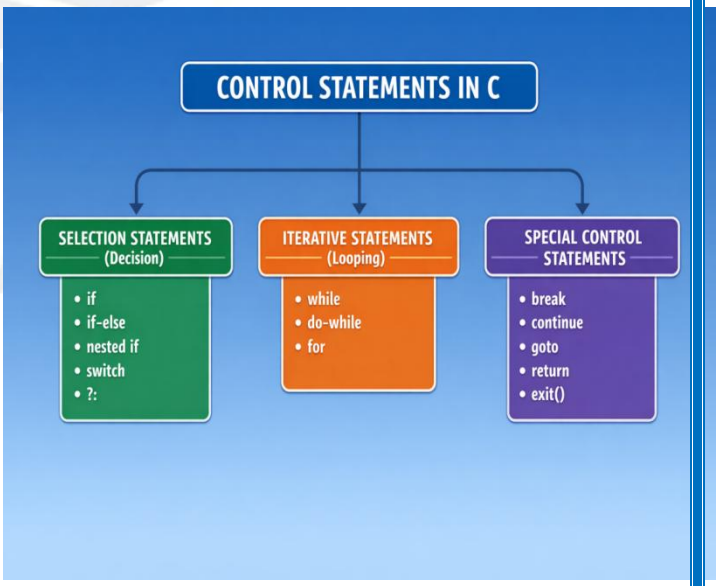
Control Statements in C

Control statements in C are used to **control the flow of execution of a program**. Normally, a C program executes statements sequentially, one after another. Control statements allow the program to **make decisions, repeat tasks, or jump to different parts of the program** based on certain conditions. They are essential for implementing logic, decision-making, and repetition in programs.

Classification of Control Statements

Control statements in C are broadly classified into **three categories**:

1. **Selection (Decision-Making) Statements**
2. **Iterative (Looping) Statements**
3. **Special Control (Jump) Statements**



1. Selection (Decision-Making) Statements

These statements are used to **select one block of code for execution** based on a condition.

- `if`
- `if-else`
- **Nested** `if`
- `switch`
- **Conditional (Ternary) Operator** `?:`

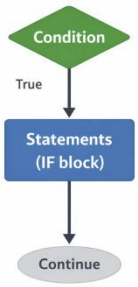
`if` Statement

The `if` statement is a **selection (decision-making) control statement** in C used to **execute a block of code only when a given condition is true**. It allows the program to make decisions

and control the flow of execution based on logical expressions or relational conditions. If the condition evaluates to true (non-zero), the statements inside the `if` block are executed; if the condition is false (zero), the block is skipped and the program continues with the next statement.

Syntax

```
if (condition) { // statements }
```



Example

```
#include <stdio.h>

int main() {
   int marks = 60;

   if (marks >= 40) {
      printf("Student is Passed");
   }

   return 0;
}
```

Explanation:

In this example, the condition `marks >= 40` is checked. Since the condition is true, the message **"Student is Passed"** is printed. If the marks were less than 40, nothing would be printed.

Key Points

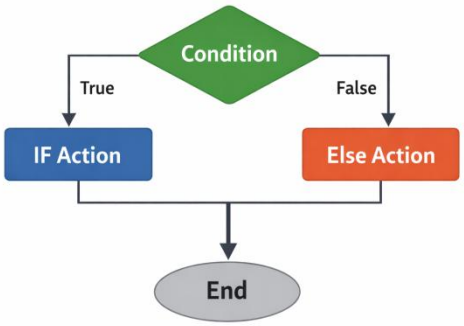
- Executes statements **only when the condition is true**
- Condition must be enclosed in parentheses `( )`
- Used for simple decision making
- Can be combined with relational and logical operators

if–else Statement

The `if-else` statement is a **selection (decision-making) control statement** in C used to **execute one block of code when a condition is true and another block when the condition is false**. It provides a clear choice between two alternative paths of execution, ensuring that **exactly one block** is executed based on the result of the condition.

Syntax

```
if (condition)
{
   // statements executed when condition is true
} else {
   // statements executed when condition is false
}
```



Example

```
#include <stdio.h>

int main() {
   int marks = 35;

   if (marks >= 40) {
      printf("Student is Passed");
   } else {
      printf("Student is Failed");
   }

   return 0;
}
```

Explanation:

The condition `marks >= 40` is checked. Since the condition is false, the `else` block is executed and **"Student is Failed"** is displayed. If the marks were 40 or above, the `if` block would be executed instead.

- Executes **one of two blocks**
- Improves clarity in decision making
- Condition must be enclosed in parentheses
- Commonly used for pass/fail, even/odd, maximum/minimum logic

Nested `if`

A **nested `if` statement** is an `if` statement placed **inside another `if` or `else` block**. It is used when a program needs to **check multiple conditions in a hierarchical manner**, where the inner condition is evaluated **only if** the outer condition is true. This structure is useful for complex decision-making, such as grading systems, eligibility checks, or multi-level validations.

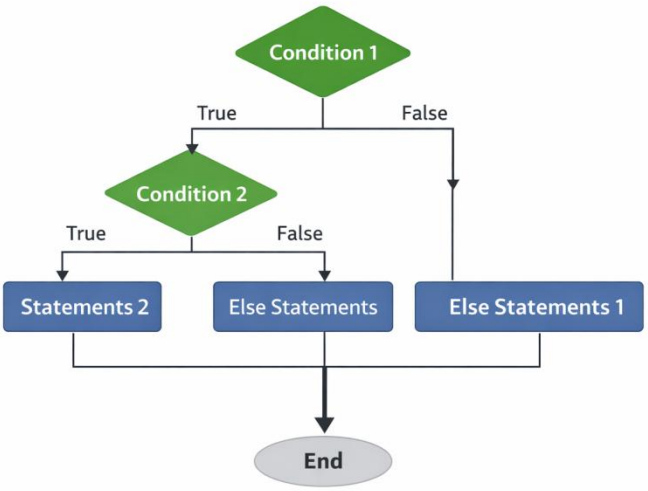
Syntax

```
if (condition1) {
   if (condition2) {
```

```
        // statements when both condition1 and condition2 are true
    }
}
```

With else:

```
if (condition1) {
    if (condition2) {
        // statements
    } else {
        // statements
    }
} else {
    // statements
}
```



Example

```
#include <stdio.h>

int main() {
    int marks = 82;

    if (marks >= 40) {
        if (marks >= 75) {
            printf("Distinction");
        } else {
            printf("Pass");
        }
    } else {
        printf("Fail");
    }

    return 0;
}
```

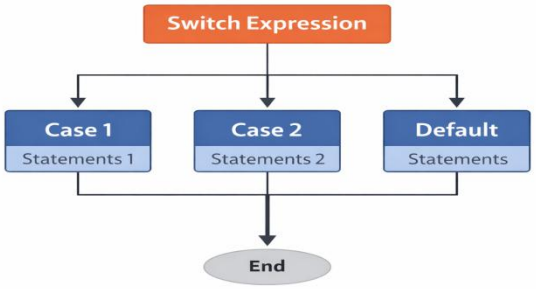
Explanation:

- First, the program checks whether `marks >= 40`.
- If true, it checks the second condition `marks >= 75`.
- If both are true, **"Distinction"** is printed.
- If the first is true but the second is false, **"Pass"** is printed.
- If the first condition is false, **"Fail"** is printed.

Key Points

- Used for **multi-level decision making**
- Inner `if` executes only if the outer `if` is true
- Can become complex if overused—proper indentation is important
- Often replaceable by `else if` for better readability

switch



The `switch` statement is a **selection (decision-making) control statement** in C used to **execute one block of code from multiple options** based on the value of an expression. It is an alternative to long `if-else if` ladders and is especially useful when a variable is compared against **constant values**. The expression in the `switch` statement is evaluated once, and control is transferred to the matching `case` label. If no case matches, the `default` block is executed.

Syntax

```
switch (expression)
{
   case constant1:
      // statements
      break;

   case constant2:
      // statements
      break;

   default:
      // statements
}
```

Important Rules

- The expression must be of **integer or character type**
- case labels must be **constant values**

- break is used to prevent **fall-through**
- default is optional but recommended

Example

```
#include <stdio.h>

int main() {
    int choice = 2;

    switch (choice) {
        case 1:
            printf("Addition");
            break;
        case 2:
            printf("Subtraction");
            break;
        case 3:
            printf("Multiplication");
            break;
        default:
            printf("Invalid choice");
    }

    return 0;
}
```

Explanation:

- The value of `choice` is evaluated.
- Since `choice` is `2`, the statements under `case 2` are executed.
- The `break` statement stops further execution and exits the `switch` block.
- If no case matched, the `default` block would execute.

Advantages

- Improves program readability
- Easy to maintain compared to long `if-else` ladders
- Faster execution in some cases

Limitations

- Works only with constant values
- Cannot use relational or logical expressions in `case`

- Cannot handle ranges directly

Conditional (Ternary) Operator ?:

The **conditional (ternary) operator** `?:` is a **selection control operator** in C used as a **compact alternative to the** `if-else` **statement**. It evaluates a condition and returns one of two values depending on whether the condition is true or false. Since it uses **three operands**, it is called the *ternary* operator. This operator is mainly used for **simple decision-making expressions** where concise code is preferred.

Syntax condition ? expression1 : expression2;

- ***If condition is true, expression1 is executed***
- ***If condition is false, expression2 is executed***

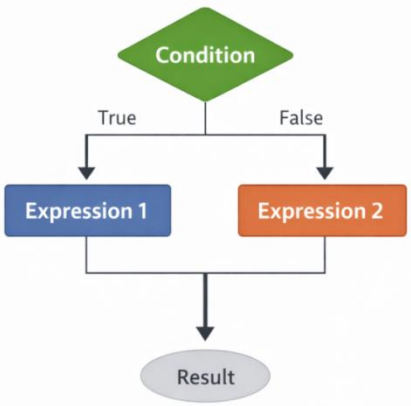
Example

```
#include <stdio.h>

int main() {
    int a = 10, b = 20;
    int max;

    max = (a > b) ? a : b;
    printf("Maximum value = %d", max);

    return 0;
}
```

***Explanation*:**
The condition `(a > b)` is evaluated. Since it is false, the value of `b` is assigned to `max`. The program then prints the maximum value.

Key Points

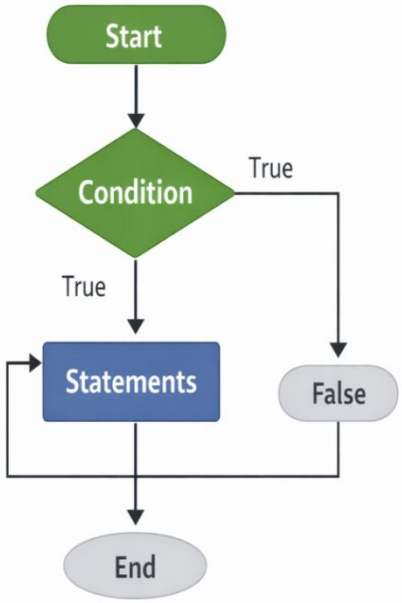
- Called **ternary** because it uses **three operands**
- Works as a **replacement for simple if–else**
- Both expressions should return **compatible data types**



- Mainly used for **short and simple decisions**
- Improves **code compactness**

Limitations

- Not suitable for complex logic
- Overuse may reduce code readability
- Cannot replace all `if-else` statements



2. Iterative (Looping) Statements

These statements are used to **repeat a block of code** as long as a given condition is satisfied.

- `while`
- `do-while`
- `for`

while Loop

The **`while` loop** is an **iterative (looping) control statement** in C used to **execute a block of statements repeatedly as long as a given condition is true**. It is called an **entry-controlled loop** because the condition is checked **before** the loop body is executed.

Syntax

```
while (condition)
{
   // statements
}
```

1. The condition is evaluated first
2. If the condition is **true**, the loop body executes
3. After execution, control returns to the condition
4. The loop continues until the condition becomes **false**

Example

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

```
int main() {
    int i = 1;

    while (i <= 5) {
        printf("%d ", i);
        i++;
    }

    return 0;
}
```

Output:

```
1 2 3 4 5
```

Key Points

- Entry-controlled loop
- Condition must be updated inside the loop
- If condition is false initially, loop body **may not execute**
- Used when number of iterations is **not known in advance**

Advantages

- Simple to use
- Suitable when repetitions depend on conditions
- Flexible loop control

Limitations

- Infinite loop if condition is not updated
- Not ideal when iteration count is fixed

do–while Loop

The **`do-while` loop** is an **iterative (looping) control statement** in C used to **execute a block of statements repeatedly until a given condition becomes false**. It is called an **exit-controlled loop** because the condition is checked **after** the loop body is executed. Hence, the loop body executes **at least once**, even if the condition is false initially.

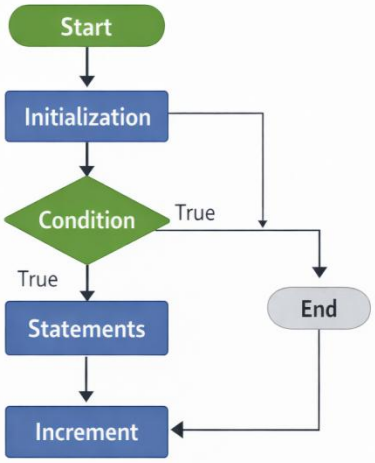
Syntax

```
do {
    // statements
```

} while (condition);

Note: Semicolon (`;`) at the end is compulsory.

Working



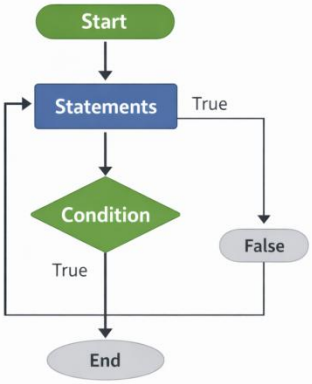
1. The loop body executes first
2. The condition is evaluated after execution
3. If the condition is **true**, the loop repeats
4. If the condition is **false**, the loop terminates

for Loop

The **`for` loop** is an **iterative (looping) control statement** in C used to **execute a block of statements a fixed number of times**. It is an **entry-controlled loop**, where the loop condition is checked **before** executing the loop body. The `for` loop is especially suitable when the **number of iterations is known in advance**.

Syntax

```
for (initialization; condition; increment/decrement)
{
   // statements
}
```



Components of `for` Loop

1. **Initialization** – Initializes the loop control variable
2. **Condition** – Checked before each iteration
3. **Increment / Decrement** – Updates the loop variable after each iteration

Working

1. Initialization is executed once
2. Condition is checked
3. If condition is **true**, loop body executes
4. Increment/decrement is applied

5. Control goes back to condition
6. Loop ends when condition becomes **false**

Example

```
#include <stdio.h>

int main() {
   int i;

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

   return 0;
}
```

Output:

```
1 2 3 4 5
```

Key Points

- Entry-controlled loop
- All loop components written in one line
- Initialization, condition, and increment are optional
- Suitable when number of repetitions is known

Advantages

- Compact and clean structure
- Easy to understand and maintain
- Ideal for counting loops

Limitations

- Less flexible when conditions are complex
- Infinite loop possible if condition is incorrect

Comparison Table: All Loops in C (`while`, `do-while`, `for`)

Aspect	*while Loop*	*do–while Loop*	*for Loop*
Loop Type	*Iterative*	*Iterative*	*Iterative*
Control Type	***Entry-controlled***	***Exit-controlled***	***Entry-controlled***
Condition Check	*Before loop body*	*After loop body*	*Before loop body*
Minimum Executions	*0 times*	***At least 1 time***	*0 times*
Syntax Simplicity	*Simple*	*Simple (needs ;)*	*Compact (all in one*

			line)
Initialization	*Before loop*	*Before loop*	*Inside for statement*
Increment/Decrement	*Inside loop body*	*Inside loop body*	*Inside for statement*
Best Used When	*Iterations not known*	*Must execute once*	*Iterations known*
Risk of Infinite Loop	*If condition not updated*	*If condition not updated*	*If condition/update wrong*
Readability	*Moderate*	*Moderate*	*High*
Common Use	*Condition-based repetition*	*Menu-driven programs*	*Counting loops*

3. Special Control (Jump) Statements

These statements are used to **alter the normal flow of execution** of the program.

- ***break***
- ***continue***
- ***goto***
- ***return***
- ***exit()***

break

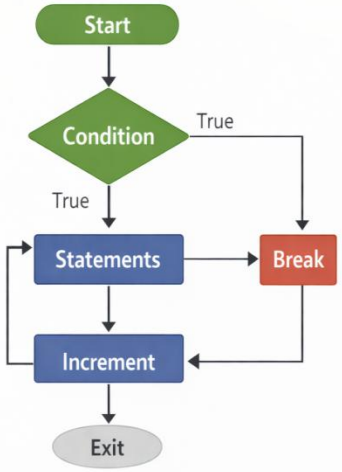
The **`break` statement** is a **special control (jump) statement** in C used to **terminate the execution of a loop or a `switch` statement immediately**. When `break` is encountered, control is transferred to the statement that follows the loop or `switch`, skipping the remaining statements inside it.

Usage of `break`

- Used inside **loops** (`for`, `while`, `do-while`)
- Used inside **`switch` statements**
- Cannot be used outside loops or switch blocks

Working

1. Program enters a loop or `switch`
2. When `break` is executed, the loop or switch **terminates immediately**
3. Control moves to the next statement after the loop or switch



Example: `break` in Loop

```
#include <stdio.h>

int main() {
    int i;

    for (i = 1; i <= 10; i++) {
        if (i == 5)
            break;
        printf("%d ", i);
    }

    return 0;
}
```

Output:

```
1 2 3 4
```

Example: break in switch

```
switch (choice)
{
case 1: printf("One");
break;
case 2:
 printf("Two");
break;
default: printf("Invalid");
}
```

Key Points

- Terminates loop or switch immediately
- Control jumps outside the structure
- Prevents fall-through in `switch`
- Only exits the **nearest enclosing loop**

Advantages

- Allows early termination
- Improves efficiency
- Useful for search operations

continue Statement

The `continue` **statement** is a **special control (jump) statement** in C used to **skip the remaining statements of the current loop iteration** and transfer control to the **next iteration of the loop**. Unlike `break`, it does not terminate the loop; it only skips the current cycle.

Usage of `continue`

- Used only inside loops (`for`, `while`, `do-while`)
- Not used with `switch` statements
- Applies to the **nearest enclosing loop**

Working

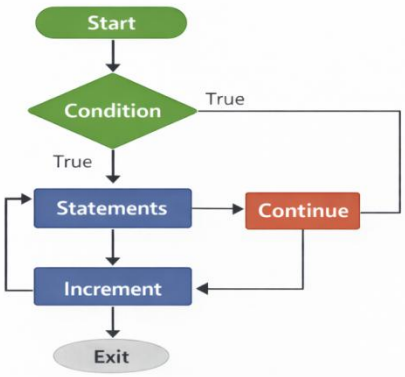
- When `continue` is executed:
- In a `for` loop → control goes to the increment/decrement step
- In `while` / `do-while` → control goes to the condition check

Example

```
#include <stdio.h>

int main()
{
   int i;

   for (i = 1; i <= 5; i++) {
      if (i == 3)
         continue;
      printf("%d ", i);
   }

   return 0;
}
```

Output:

1 2 4 5



Key Points

- Skips current iteration only
- Loop continues normally

- Does not terminate the loop
- Useful for skipping unwanted values

Aspect	*break*	*continue*
Type	*Jump (control) statement*	*Jump (control) statement*
Purpose	*Terminates the loop or switch*	*Skips the current iteration of a loop*
Effect on loop	*Loop ends immediately*	*Loop continues*
Iterations	*All remaining iterations are stopped*	*Only current iteration is skipped*
Usage in loops	*Yes*	*Yes*
Usage in switch	*Yes*	*No*
Control transfer	*Moves control outside the loop/switch*	*Moves control to next iteration*
Behavior in for loop	*Exits loop completely*	*Jumps to increment/decrement part*
Behavior in while / do–while	*Exits loop completely*	*Jumps to condition checking*
Typical use case	*When loop must stop (e.g., value found)*	*When an iteration must be ignored*
Risk if misused	*Premature loop termination*	*Infinite loop (if update skipped)*

`goto` Statement

The **`goto` statement** is a **special control (jump) statement** in C used to **transfer control unconditionally** from one part of the program to another **within the same function**. It jumps to a **labeled statement** specified by the programmer.

Syntax

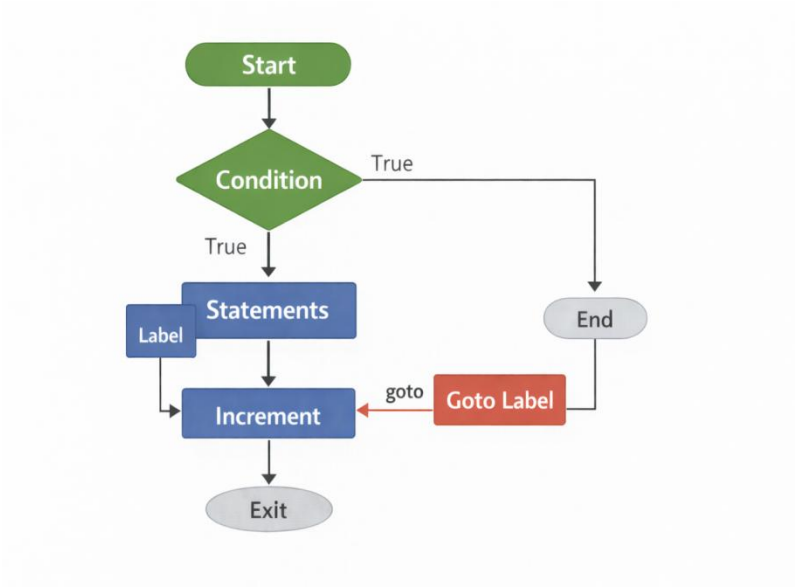
```
goto label;
/* statements */
label:
    // statements
```

Working

1. When `goto label;` is encountered, control **jumps directly** to the statement marked with `label:`
2. The jump is **unconditional** (no condition check)
3. Execution continues from the labeled statement

Example

```
#include <stdio.h>

int main() {
    int i = 1;

start:
    printf("%d ", i);
    i++;

    if (i <= 5)
        goto start;

    return 0;
}
```



Output:

```
1 2 3 4 5
```

Use Cases

- Breaking out of **deeply nested loops**
- Error handling and cleanup (rare cases)
- Jumping to a common exit point

Advantages

- Simple to implement jumps
- Useful in limited, controlled situations
- Reduces duplicate code in error handling

Disadvantages

- Makes programs hard to read and maintain
- Can lead to **spaghetti code**
- Violates structured programming principles

return Statement

The **`return` statement** is a **control (jump) statement** used to **terminate the execution of a function** and **transfer control back to the calling function**. It may optionally return a value to the caller.

Syntax

```
return;         // for functions with void return type
return value;   // for functions that return a value
```

Working

- When `return` is executed:
  - The current function **stops execution**
  - Control goes back to the **calling function**
  - If a value is specified, it is **returned to the caller**

Example

```
#include <stdio.h>

int add(int a, int b) {
    return a + b;
}

int main() {
    int sum = add(10, 20);
    printf("Sum = %d", sum);
    return 0;
}
```

Explanation:

`return a + b;` sends the result back to `main()`.
`return 0;` ends the `main()` function normally.

- Used to exit a **function**
- Can return a value
- Commonly used in `main()` as `return 0;`
- Control goes back to **calling function**

exit() Function

The **`exit()` function** is a **library function** used to **terminate the entire program immediately**, regardless of where it is called.

Header File

```
#include <stdlib.h>
```

Syntax

exit(status);

- `status = 0` → *normal termination*
- `status ≠ 0` → *abnormal termination*

Working

- When `exit()` is called:
  - **All functions stop execution**
  - Program terminates immediately
  - Control returns to the **operating system**
  - Open files are closed automatically

Example

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

int main() {
   printf("Before exit\n");
   exit(0);
   printf("After exit");   // This will not execute
}
```

Output:

Before exit

Arrays in C

An **array** in C is a **collection of elements of the same data type** stored in **contiguous memory locations** and accessed using a **common name with an index (subscript)**. Arrays are used to store and process multiple values efficiently using a single variable name.

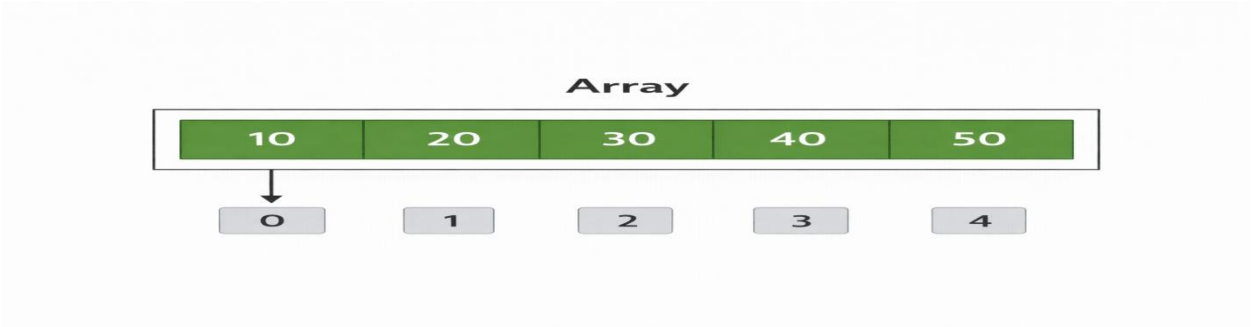
- To store **multiple values** of the same type
- To reduce the number of variables
- To simplify program logic using loops
- To handle large data sets efficiently

Types of Arrays in C

1. **One-Dimensional Array**
2. **Multidimensional Array** (Two-dimensional, etc.)

Declaration of Array in C

The **declaration of an array** in C specifies the **data type**, **name of the array**, and the **number of elements (size)** that the array can store. It informs the compiler about the memory required to store the array elements.



General Syntax

data_type array_name[size];

- **data_type** → Type of elements (int, float, char, etc.)
- **array_name** → Name of the array
- **size** → Number of elements (must be a constant integer)

Examples of Array Declaration

1. Integer Array

```
int marks[5];
```

Declares an array named `marks` that can store **5 integer values**.

Initialization of Array in C

Array initialization in C can be done in **different ways** depending on how and when the values are assigned to the array elements. The following are the **main types of array initialization** commonly asked in exams.

1. Compile-Time Initialization

In compile-time initialization, the values of the array elements are **assigned at the time of declaration**.

Example

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

2. Partial Initialization

When only **some elements are initialized**, the remaining elements are automatically initialized to **zero**.

Example

```
int a[5] = {1, 2};
```

Result:

```
1 2 0 0 0
```

3. Initialization Without Size

The array size is **automatically determined** by the compiler based on the number of values provided.

Example

```
int a[] = {5, 10, 15, 20};
```

4. Run-Time Initialization

In run-time initialization, array elements are **assigned values during program execution**, usually using loops or input functions.

Example

```
int a[3], i;

for (i = 0; i < 3; i++) {
    scanf("%d", &a[i]);
}
```

5. Character Array (String) Initialization

Character arrays can be initialized using **string literals**. The compiler automatically adds the null character (`'\0'`).

Example

```
char name[] = "C Programming";
```

6. Multidimensional Array Initialization

Multidimensional arrays are initialized using **nested braces**, where each inner brace represents a row.

Example

```
int mat[2][2] = {{1, 2}, {3, 4}};
```

One-Dimensional Array (1D Array)

A **one-dimensional array (1D array)** in C is a collection of elements of the **same data type** stored in **contiguous memory locations** and accessed using a **single index (subscript)**. It represents data in a **linear or sequential form**, similar to a list. Each element in the array is identified by its position, starting from index **0** to **size − 1**. One-dimensional arrays are commonly used to store multiple related values, such as marks of students, salaries of employees, or a list of numbers, using a single variable name.

Syntax

```
data_type array_name[size];
```

Example:

```
#include <stdio.h>

int main() {
   int marks[5] = {60, 70, 80, 90, 75};
   int i;

   for (i = 0; i < 5; i++) {
      printf("%d ", marks[i]);
   }

   return 0;
}
```

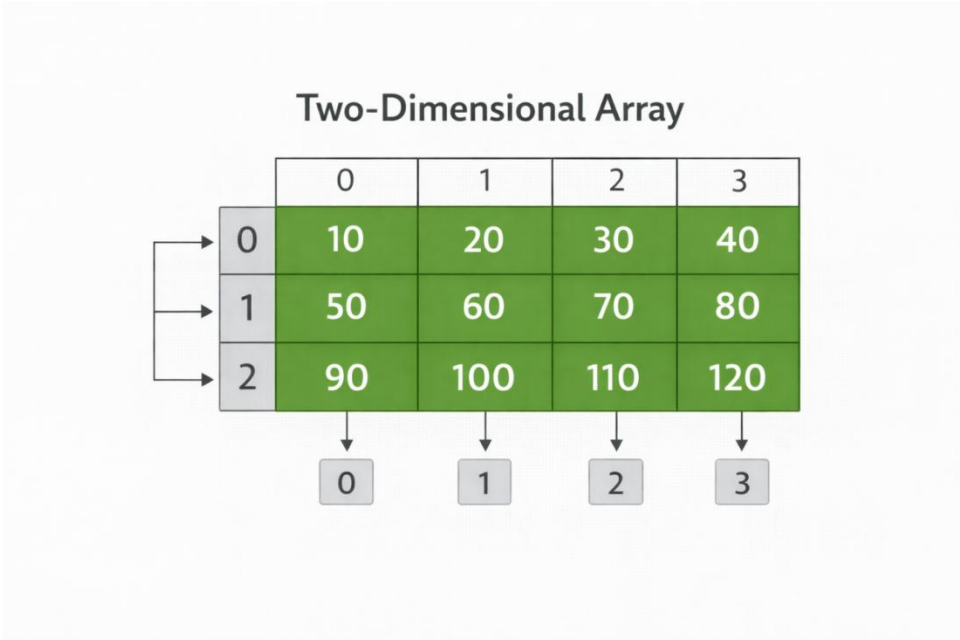
In this example, `marks` is a one-dimensional array that stores five integer values. The `for` loop accesses each element using its index and prints the values sequentially.

Two-Dimensional Array (2D Array)

A **two-dimensional array (2D array)** in C is a collection of elements of the **same data type** arranged in the form of **rows and columns**, similar to a **matrix or table**. It is accessed using **two indices**, where the first index represents the **row number** and the second index represents the **column number**. Like all arrays in C, the index values start from **0**. Two-dimensional arrays are commonly used to store tabular data such as matrices, marks of students in different subjects, or any data that naturally fits into a grid format.

Syntax

```
data_type array_name[rows][columns];
```



Example:

```
#include <stdio.h>

int main() {
   int mat[2][3] = {{1, 2, 3}, {4, 5, 6}};
   int i, j;

   for (i = 0; i < 2; i++) {
      for (j = 0; j < 3; j++) {
         printf("%d ", mat[i][j]);
      }
      printf("\n");
   }

   return 0;
}
```

In this example, `mat` is a two-dimensional array with **2 rows and 3 columns**. The nested `for` loops are used to access and display each element of the array in matrix form.

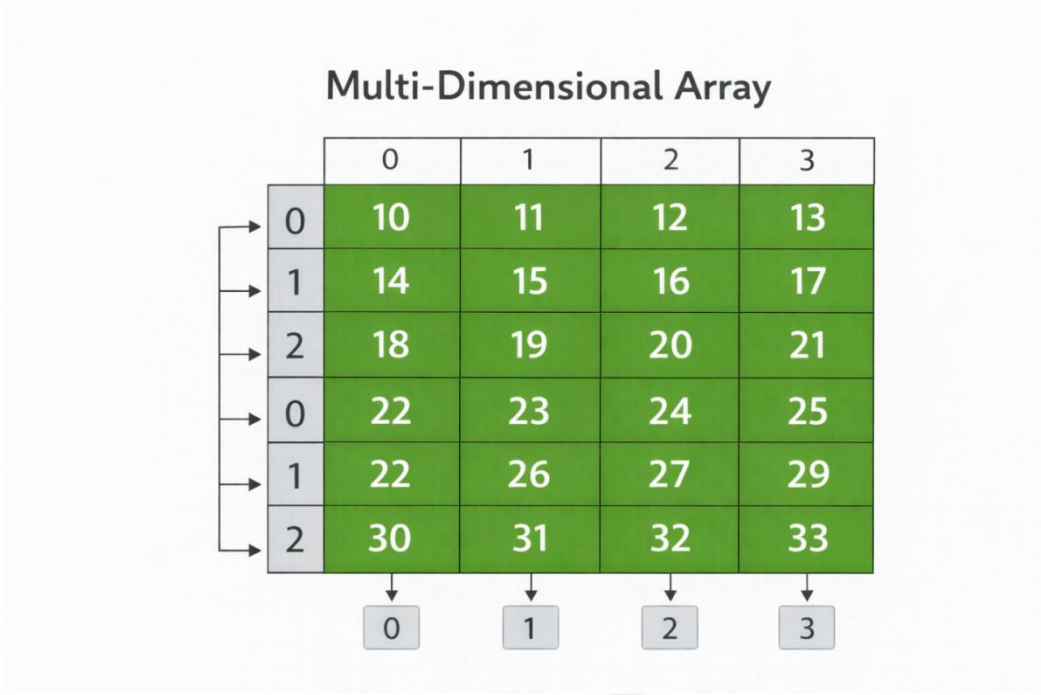
Multidimensional Array

A **multidimensional array** in C is an array that uses **more than two indices** to access its elements. In other words, it is an array of arrays of arrays. While one-dimensional arrays store data in a linear form and two-dimensional arrays store data in rows and columns, multidimensional arrays extend this concept to **three or more dimensions**. These arrays are

useful for representing **complex data structures** such as 3D matrices, scientific data, and multi-level tabular information.

Each dimension represents a level of data organization, and array indices in every dimension start from **0**.

Example (Three-Dimensional Array):




```
#include <stdio.h>
int main()
{
   int a[2][2][2] = {
      {{1, 2}, {3, 4}},
      {{5, 6}, {7, 8}}
   };
   int i, j, k;

   for (i = 0; i < 2; i++) {
      for (j = 0; j < 2; j++) {
         for (k = 0; k < 2; k++) {
            printf("%d ", a[i][j][k]);
         }
         printf("\n");
      }
      printf("\n");
   }
```

```
    return 0;
}
```

In this example, `a` is a **three-dimensional array** with dimensions 2 × 2 × 2. The three nested loops are used to access and display each element, demonstrating how multidimensional arrays store and organize data across multiple levels.

Unit- III

Meaning of a Function

A **function** in C is a **self-contained block of statements** designed to perform a **specific task**. A function takes input in the form of **parameters**, processes the data, and may return a result to the calling program. Functions help in dividing a large program into smaller, manageable units, making the program easier to understand, develop, and maintain. In C, every program starts execution from the **`main()` function**, and additional functions can be created to perform different operations.

Advantages of Functions

Functions provide several advantages in C programming:

1. **Modularity** – Programs can be divided into smaller modules, each performing a specific task.
2. **Reusability** – A function can be used multiple times in the same program or in different programs.
3. **Improved Readability** – Code becomes easier to read and understand due to logical separation.
4. **Ease of Debugging** – Errors can be easily identified and fixed within individual functions.
5. **Reduced Code Duplication** – Avoids repetition of the same code, making programs concise.
6. **Team Development** – Different programmers can work on different functions simultaneously.

Need for Functions

Functions are needed to handle **large and complex programs efficiently**. Without functions, a program would consist of a long sequence of statements, making it difficult to manage and debug. By using functions, tasks can be performed independently, changes can be made without affecting the entire program, and testing becomes simpler. Functions also promote **structured programming**, which improves program quality, maintainability, and reliability.

1. Function Declaration (Function Prototype)

A **function declaration** tells the compiler **about the function's name, return type, and parameters** before it is used. It does not contain the function body. The main purpose of a declaration is to inform the compiler so that it can perform **type checking** during function calls.

Syntax

```
return_type function_name(parameter_list);
```

Example

```
int add(int, int);
```

2. Function Definition

A **function definition** contains the **actual body of the function**, where the task is performed. It specifies **what the function does**. The function definition includes the return type, function name, parameters, and a block of executable statements.

Syntax

```
return_type function_name(parameter_list) {
    // function body
}
```

Example

```
int add(int a, int b) {
    return a + b;
}
```

3. Function Calling

A **function call** is a statement that **invokes the function** to execute its code. When a function is called, control is transferred from the calling function (usually `main()`) to the called function. After execution, control returns back to the calling function, along with the return value (if any).

Syntax

```
function_name(arguments);
```

Example

```
sum = add(10, 20);
```

Flow of Function Execution in C

The execution of a function in C follows a **well-defined sequence**. Each step plays an important role in ensuring correct program execution and communication between functions.

1. Function is Declared (Prototype)

A **function declaration**, also called a **function prototype**, informs the compiler about the **function name, return type, and parameters** before it is used. This step allows the compiler to perform **type checking** and ensures that the function is called correctly. The declaration does not contain the function body; it only specifies the function's interface.

Example:

```
int add(int, int);
```

2. Function is Called from main()

A **function call** is made from the `main()` function or any other function to execute the defined task. When the function call is encountered, the control temporarily leaves the calling function, and the **actual arguments** are passed to the function's parameters.

Example:

```
result = add(10, 20);
```

3. Control Transfers to Function Definition

After the function call, program control moves to the **function definition**, where the actual code of the function is present. The values passed during the function call are assigned to the **formal parameters** of the function.

Example:

```
int add(int a, int b) {
    // control enters here
```

4. Function Executes and Returns Value

The statements inside the function are executed sequentially. If the function has a return type other than `void`, it uses the **`return` statement** to send a value back to the calling function. Execution of the function stops once the return statement is reached.

Example:

```
return a + b;
```

5. Control Comes Back to `main()`

After the function finishes execution, control returns to the **calling point in `main()`**. The returned value is stored in the variable used in the function call, and the program continues executing the remaining statements in `main()`.

Example:

```
printf("Sum = %d", result);
```

Complete Example Program

```
#include <stdio.h>

/* Function Declaration */
int add(int, int);
int main() {
   int result;
   result = add(10, 20);  // Function Call
   printf("Sum = %d", result);

   return 0;
}
/* Function Definition */
int add(int a, int b) {
   return a + b;
}
```

1.Library Functions (Built-in Functions)

Library functions are **predefined functions** provided by the C standard library. They are already written, tested, and stored in header files. Programmers can directly use these functions by including the appropriate header file.

Examples

- `printf()` – Output function
- `scanf()` – Input function
- `strlen()` – String length
- `sqrt()` – Square root
- `pow()` – Power function

Example Program

```
#include <stdio.h>

int main() {
   printf("Welcome to C");
   return 0;
}
```

Advantages

- Saves development time
- Reliable and efficient
- Easy to use

2. User-Defined Functions

Meaning

User-defined functions are functions **created by the programmer** to perform a specific task. They help in modular programming and code reusability.

Types of User-Defined Functions

User-defined functions are further classified based on **arguments and return value**:

1. Function with **no arguments and no return value**
2. Function with **arguments and no return value**
3. Function with **no arguments and return value**
4. Function with **arguments and return value**

Example

```
#include <stdio.h>
void greet() {
   printf("Welcome to C Programming");
}
int main() {
   greet();
   return 0;
}
```

2. Function with Arguments and No Return Value

A **function with arguments and no return value** receives data from the calling function through parameters but does not return any result. The function processes the input data and displays or uses the result internally. This type of function is useful when the output is directly printed instead of being returned.

Example

```
#include <stdio.h>
void displaySum(int a, int b)
{
   printf("Sum = %d", a + b);
}
int main() {
   displaySum(10, 20);
   return 0;
}
```

3. Function with No Arguments and Return Value

A **function with no arguments and return value** does not receive any data from the calling function but returns a value after performing its task. The function itself generates the required data and sends the result back to the calling function. This type of function is useful when data generation and processing are handled within the function.

Example

```
#include <stdio.h>

int getNumber() {
    return 100;
}

int main() {
    int num;
    num = getNumber();
    printf("Number = %d", num);
    return 0;
}
```

4. Function with Arguments and Return Value

A **function with arguments and return value** receives input data from the calling function and returns a processed result. This is the **most commonly used and most flexible type** of function. It promotes modular programming, reusability, and clear data flow between functions.

Example

```
#include <stdio.h>

int add(int a, int b) {
    return a + b;
}

int main() {
    int result;
    result = add(15, 25);
    printf("Sum = %d", result);
    return 0;
}
```

CType Functions in C

Function	Purpose	Returns True When	Example
`isalpha()`	Alphabet check	Character is A–Z or a–z	`isalpha('A')`
`isdigit()`	Digit check	Character is 0–9	`isdigit('5')`

`isalnum()`	*Alphanumeric check*	*Letter or digit*	`isalnum('9')`
`islower()`	*Lowercase check*	*Lowercase letter*	`islower('a')`
`isupper()`	*Uppercase check*	*Uppercase letter*	`isupper('A')`
`isspace()`	*Whitespace check*	*Space, tab, newline*	`isspace(' ')`
`ispunct()`	*Punctuation check*	*! , . ? etc.*	`ispunct('!')`
`isxdigit()`	*Hex digit check*	*0–9, A–F, a–f*	`isxdigit('F')`
`toupper()`	*To uppercase*	*Converts lowercase*	`toupper('a') → 'A'`
`tolower()`	*To lowercase*	*Converts uppercase*	`tolower('A') → 'a'`

String Functions in C

In C, **strings are arrays of characters terminated by a null character (`'\0'`)**. The C standard library provides a set of **string handling functions** defined in the header file **`<string.h>`**. These functions are used to **manipulate, compare, copy, concatenate, and search strings** efficiently.

Header File
#include <string.h>

1. strlen() – String Length Function

The `strlen()` function is used to **find the length of a string**. It counts the number of characters in the string **excluding the null character (`'\0'`)**. This function is commonly used to determine the size of a string during processing.

Example

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

int main() {
   char str[] = "Computer";
   printf("Length = %lu", strlen(str));
   return 0;
}
```

2. strcpy() – String Copy Function

The `strcpy()` function copies the contents of one string (**source**) into another string (**destination**). The destination string must have **sufficient memory** to store the copied string including the null character.

Example

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

```
#include <string.h>
int main() {
   char src[] = "C Language";
   char dest[20];

   strcpy(dest, src);
   printf("%s", dest);
   return 0;
}
```

3. strncpy() – Limited String Copy

Explanation

The `strncpy()` function copies **only a specified number of characters** from the source string to the destination string. It is useful when partial copying is required.

Example

```
#include <stdio.h>
#include <string.h>
int main() {
   char src[] = "Programming";
   char dest[10];
   strncpy(dest, src, 6);
   dest[6] = '\0';
   printf("%s", dest);
   return 0;
}
```

4. strcat() – String Concatenation

The `strcat()` function **appends one string to the end of another string**. The destination string must have enough space to hold the combined result.

Example

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

int main() {
   char s1[20] = "Hello ";
   char s2[] = "World";

   strcat(s1, s2);
   printf("%s", s1);
```

```
    return 0;
}
```

5 strncat() – Limited Concatenation

The `strncat()` function appends **only a specified number of characters** from the source string to the destination string.

Example

```
#include <stdio.h>
#include <string.h>
int main() {
    char s1[20] = "Good ";
    char s2[] = "Morning";
    strncat(s1, s2, 4);
    printf("%s", s1);
    return 0;
}
```

6. strcmp() – String Comparison

The `strcmp()` function compares two strings **lexicographically**.

- Returns **0** if both strings are equal
- Returns **positive** value if first string is greater
- Returns **negative** value if first string is smaller

Example

```
#include <stdio.h>
#include <string.h
int main() {
   char s1[] = "Apple";
   char s2[] = "Banana";
   printf("%d", strcmp(s1, s2));
   return 0;
}
```

7. strncmp() – Limited String Comparison

The `strncmp()` function compares **only the first `n` characters** of two strings.

Example

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

```
int main() {
    char s1[] = "Computer";
    char s2[] = "Complain";
    printf("%d", strncmp(s1, s2, 3));
    return 0;
}
```

8. *strchr() – Find Character in String*

The `strchr()` function finds the **first occurrence of a character** in a string and returns a pointer to it.

Example

```
#include <stdio.h>
#include <string.h>
int main() {
    char str[] = "Language";
    char *p = strchr(str, 'g');
    printf("%s", p);
    return 0;
}
```

9. strstr() – Find Substring

Explanation

The `strstr()` function searches for a **substring** within a string and returns a pointer to the first occurrence.

Example

```
#include <stdio.h>
#include <string.h>
int main() {
    char str[] = "C Programming Language";
    char *p = strstr(str, "Programming");
    printf("%s", p);
    return 0;
}
```

10. *strrev() – Reverse String* (Compiler Dependent)

The `strrev()` function reverses the characters of a string. This function is **not part of ANSI C**, but available in some compilers.

Example

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

String Functions in C

Function	Purpose	Syntax	Example
strlen()	Finds length of string (excluding \0)	strlen(str)	strlen("CSE") → 3
strcpy()	Copies source string to destination	strcpy(dest, src)	strcpy(d, s);
strncpy()	Copies first n characters	strncpy(dest, src, n)	strncpy(d,s,4);
strcat()	Appends one string to another	strcat(dest, src)	strcat(s1,s2);
strncat()	Appends first n characters	strncat(dest, src, n)	strncat(s1,s2,3);
strcmp()	Compares two strings	strcmp(s1, s2)	strcmp("A","B")
strncmp()	Compares first n characters	strncmp(s1, s2, n)	strncmp(s1,s2,2);
strchr()	Finds first occurrence of character	strchr(str, ch)	strchr("Hello",'e')
strstr()	Finds first occurrence of substring	strstr(str, sub)	strstr("C Prog","Prog")
strrev()*	Reverses string	strrev(str)	strrev("CSE")

Call by Value vs Call by Reference (C Programming)

In C, when a function is called, values can be passed to it in **two ways**: **Call by Value** and **Call by Reference**. These methods determine **how data is transferred** between the calling function and the called function.

---

1. Call by Value

Explanation

In **call by value**, a **copy of the actual argument** is passed to the function. The function works on this copy, not on the original variable. Therefore, **any changes made inside the**

function do not affect the original variable in the calling function. This method ensures data safety but does not allow the function to modify the original values.

Example

```
#include <stdio.h>


void change(int x) {
   x = 20;
}


int main() {
   int a = 10;
   change(a);
   printf("Value of a = %d", a);
   return 0;
}
```

Output: Value of a = 10

2. Call by Reference

Explanation

In **call by reference**, the **address of the variable** is passed to the function. The function accesses the original variable using pointers. Hence, **any changes made inside the function directly affect the original variable** in the calling function. This method is useful when we want a function to modify more than one value or work efficiently with large data.

Example

```
#include <stdio.h>


void change(int *x) {
    *x = 20;
}
```

```
int main() {
    int a = 10;
    change(&a);
    printf("Value of a = %d", a);
    return 0;
}
```

Output: Value of a = 20

Explanation:
Here, the address of `a` is passed to the function. Using pointer dereferencing (`*x`), the function modifies the original variable.

Comparison Table: Call by Value vs Call by Reference (Exam-Oriented)

Aspect	Call by Value	Call by Reference
Meaning	Copy of actual value is passed	Address of variable is passed
Data passed to function	Value	Memory address
Effect on original variable	No change	Original variable is modified
Use of pointers	Not used	Used
Memory efficiency	Less efficient (copy created)	More efficient
Data safety	High (original data safe)	Low (data can be changed)
Ability to return multiple values	Not possible	Possible
Changes inside function	Affects only local copy	Affects original variable
Default in C	Yes	Implemented using pointers
Example call	`func(a);`	`func(&a);`
Typical use case	When data should not change	When modification is required

Passing Arrays to Functions in C

Explanation

In C programming, **arrays can be passed to functions** to allow the function to process multiple elements efficiently. When an array is passed to a function, **the base address of the array is passed**, not a copy of the entire array. Therefore, any changes made to the array elements inside the function **affect the original array** in the calling function. This behavior is similar to **call by reference**.

Passing arrays to functions helps in **code reusability**, **modular programming**, and **efficient handling of large data sets** such as lists, tables, or matrices.

Syntax

```
return_type function_name(data_type array_name[], int size);
```

return_type function_name(data_type *array_name, int size);

Example: Passing One-Dimensional Array

```
#include <stdio.h>


void display(int a[], int n) {
   int i;
   for (i = 0; i < n; i++) {
      printf("%d ", a[i]);
   }
}


int main() {
   int arr[5] = {10, 20, 30, 40, 50};
   display(arr, 5);
   return 0;
}
```

Explanation:
Here, the array `arr` is passed to the function `display()`. The function receives the base address of the array and prints all elements using a loop.

Example: Modifying Array Elements Inside Function

```
#include <stdio.h>


void update(int a[], int n) {
   int i;
   for (i = 0; i < n; i++) {
      a[i] = a[i] + 5;
   }
```

```
}


int main() {
   int arr[3] = {10, 20, 30};
   update(arr, 3);


   for (int i = 0; i < 3; i++) {
      printf("%d ", arr[i]);
   }
   return 0;
} Output: 15 25 35
```

Recursion in C

Definition

Recursion is a programming technique in which a **function calls itself** to solve a problem. A recursive function breaks a complex problem into **smaller sub-problems** of the same type and solves them repeatedly until a **base condition** is met. In C, recursion is commonly used for problems that can be naturally divided into similar sub-tasks, such as factorial calculation, Fibonacci series, and tree traversal.

Key Components of Recursion

1. **Recursive Call**
   The function calling itself.
2. **Base Condition**
   A condition that stops further recursive calls.
   Without a base condition, recursion leads to **infinite calls** and program crash.

General Syntax of Recursive Function

```
return_type function_name(parameters) {
   if (base_condition)
```

```
    return value;
  else
    return function_name(modified_parameters);
}
```

Example: Factorial Using Recursion

```
#include <stdio.h>

int factorial(int n) {
  if (n == 0)      // base condition
    return 1;
  else
    return n * factorial(n - 1);  // recursive call
}

int main() {
  int result;
  result = factorial(5);
  printf("Factorial = %d", result);
  return 0;
}
```

Output: Factorial = 120

Explanation of the Example

- `factorial(5)` calls `factorial(4)`
- `factorial(4)` calls `factorial(3)`
- This continues until `factorial(0)` is reached
- When the base condition is satisfied, values are returned step by step
- Final result is calculated as `5 × 4 × 3 × 2 × 1`

Advantages of Recursion

- Simplifies complex problems
- Reduces code length
- Improves readability for mathematical problems
- Useful in divide-and-conquer algorithms

Disadvantages of Recursion

- Uses more memory (function call stack)
- Slower than iteration in some cases
- Risk of stack overflow if base condition is missing

Inline Functions in C

Definition

An **inline function** is a function in which the **function call is replaced by the actual function code** at the point of calling during compilation. The purpose of an inline function is to **reduce function call overhead** and improve execution speed, especially for **small and frequently used functions**. Inline functions are defined using the **`inline` keyword**.

Syntax

inline `return_type function_name(parameters) { // function body }`

Working of Inline Functions

- When an inline function is called, the compiler **inserts the function code directly** instead of performing a normal function call.
- This avoids the overhead of **stack operations and control transfer**.
- Inline expansion is a **request to the compiler**, not a command. The compiler may ignore it in some cases.

Example

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

```
inline int square(int x) {
    return x * x;
}

int main() {
    int result;
    result = square(5);
    printf("Square = %d", result);
    return 0;
} Output: Square = 25
```

Advantages of Inline Functions

- Faster execution
- Reduces function call overhead
- Improves performance for small functions
- Code readability is maintained

Limitations of Inline Functions

- Increases program size (code duplication)
- Not suitable for large or complex functions
- Compiler may ignore inline request
- Not effective for recursive functions

Scope of Variables

Meaning

The **scope of a variable** refers to the **region of the program where the variable is accessible or visible**. It determines **where a variable can be used** within a program. Scope helps avoid naming conflicts and controls access to data.

Types of Scope in C (List – Exam-Oriented)

In C programming, the **scope of a variable** defines where the variable can be accessed in a program. The **types of scope** are:

1. **Local Scope**
2. **Global Scope**

1. Local Scope

Explanation

A variable declared **inside a function** has **local scope**. It can be accessed **only within that function** and is not visible outside it. Local variables help prevent accidental modification of data by other functions.

Example

```
#include <stdio.h>


void show() {
   int x = 10;   // local variable
   printf("%d", x);
}


int main() {
   show();
   return 0;
}
```

2. Global Scope

Explanation

A variable declared **outside all functions** has **global scope**. It can be accessed by **any function** in the program. Global variables are useful when multiple functions need to share the same data.

Example

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

```
int x = 20;   // global variable


void display() {
   printf("%d", x);
}


int main() {
   display();
   return 0;
}
```

Storage Classes in C

Meaning of Storage Classes

Storage classes in C define the **scope, lifetime, storage location, and default initial value** of variables and functions. They control **how long a variable exists in memory**, **where it is stored**, and **who can access it**. Storage classes play a crucial role in memory management and program organization.

Types of Storage Classes in C

C supports **four main storage classes**:

1. `auto`
2. `register`
3. `static`
4. `extern`

---

1. `auto` Storage Class

Explanation

The `auto` storage class is the **default storage class for local variables**. Variables declared inside a function without any storage class keyword are automatically treated as `auto`. These variables are stored in **stack memory**, have **local scope**, and exist only during the execution of the function.

Example

```
#include <stdio.h>


int main() {
   auto int x = 10;   // auto variable
   printf("%d", x);
   return 0;
}
```

2. `register` Storage Class

Explanation

The `register` storage class is used to request the compiler to store the variable in a **CPU register** instead of memory for faster access. It is mainly used for **frequently accessed variables** such as loop counters. The address of a register variable **cannot be accessed** using the `&` operator.

Example

```
#include <stdio.h>


int main() {
   register int i;
   for (i = 0; i < 5; i++) {
      printf("%d ", i);
   }
   return 0;
}
```

3. `static` Storage Class

Explanation

The `static` storage class preserves the **value of a variable between function calls**. A static variable is initialized only once and retains its value throughout the program execution. It has **local scope but static lifetime**.

Example

```
#include <stdio.h>


void count() {
   static int x = 0;
   x++;
   printf("%d ", x);
}


int main() {
   count();
   count();
   count();
   return 0;
}
```

4. `extern` Storage Class

Explanation

The `extern` storage class is used to **declare a global variable that is defined in another file or later in the program**. It does not allocate memory but tells the compiler that the variable exists elsewhere. It is mainly used for **sharing global variables across multiple files**.

Example

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

```
extern int x;


int x = 50;


int main() {
   printf("%d", x);
   return 0;
}
```

Comparison Table of Storage Classes (Exam-Oriented)

Storage Class	Scope	Lifetime	Storage Location	Default Value
auto	Local	Function	Stack	Garbage
register	Local	Function	CPU Register	Garbage
static	Local / Global	Program	Static Memory	0
extern	Global	Program	Static Memory	0

Introduction to Pointers in C

Meaning of Pointer

A **pointer** in C is a **variable that stores the memory address of another variable**. Instead of holding a data value directly, a pointer holds the **location (address)** where the data is stored in memory. Pointers provide a powerful way to access and manipulate data indirectly and are widely used for **dynamic memory allocation, array handling, function arguments, and efficient memory management**.

Pointers are needed in C for the following reasons:

- To **access and modify variables indirectly**
- To implement **call by reference**
- To pass **arrays and strings to functions**
- To perform **dynamic memory allocation**
- To improve **program efficiency and flexibility**

Declaration of Pointer

Syntax

```
data_type *pointer_name;
```

Example

```
int *p;
```

Initialization of Pointer

A pointer must be initialized with the **address of a variable** using the **address-of operator (&)**.

```
int x = 10;
int *p = &x;
```

Accessing Value Using Pointer (Dereferencing)

The **dereference operator (*)** is used to access the value stored at the address held by the pointer.

```
printf("%d", *p);   // prints value of x
```

```
#include <stdio.h>


int main() {
   int x = 25;
   int *p;


   p = &x;


   printf("Value of x = %d\n", x);
   printf("Address of x = %p\n", &x);
   printf("Value using pointer = %d\n", *p);


   return 0;
}
```

Advantages of Pointers

- Efficient memory usage
- Enables dynamic memory allocation
- Supports advanced data structures
- Enables call by reference

Pointer Types in C

In C programming, **pointer types** specify the **type of data whose address the pointer can store**. The data type of a pointer determines **how many bytes are accessed** and **how the value is interpreted** when the pointer is dereferenced. Proper use of pointer types ensures type safety and correct memory access.

Four Special Pointer Types in C

In C programming, apart from normal pointers, there are **four special pointer types** that are very important from an **exam and conceptual point of view**.

---

1. NULL Pointer

Explanation

A **NULL pointer** is a pointer that **does not point to any valid memory location**. It is explicitly assigned the value `NULL`. Using NULL pointers helps in avoiding accidental access to unknown memory locations.

Example

```
int *p = NULL;


if (p == NULL) {
   printf("Pointer is NULL");
}
```

2. Void Pointer (Generic Pointer)

Explanation

A **void pointer** can store the address of **any data type**. It does not have a specific type, so it **cannot be dereferenced directly**. Type casting is required before accessing the value.

Example

```
int x = 10;

void *p = &x;


printf("%d", *(int *)p);
```

3. Dangling Pointer

Explanation

A **dangling pointer** is a pointer that **points to a memory location that has already been freed or deleted**. Accessing such a pointer leads to undefined behavior.

```
int *p = (int *)malloc(sizeof(int));

free(p);   // memory freed

// p is now a dangling pointer
```

4. Wild Pointer

Explanation

A **wild pointer** is a pointer that is **declared but not initialized**. It contains a garbage address and may point to any random memory location.

Example

```
int *p;   // wild pointer

*p = 10;  // dangerous
```

Pointer Arithmetic in C

Meaning

Pointer arithmetic refers to the operations performed on pointers to **move through memory locations**. Since pointers store memory addresses, arithmetic operations on pointers allow access to **different elements of an array or contiguous memory block**. Pointer arithmetic depends on the **data type of the pointer**, because each data type occupies a specific number of bytes in memory.

Allowed Pointer Arithmetic Operations

In C, only the following operations are allowed on pointers:

1. **Increment (++)**
2. **Decrement (--)**
3. **Addition with integer (+)**
4. **Subtraction with integer (-)**
5. **Difference between two pointers (-)**

❌ Addition of two pointers, multiplication, division, or modulus on pointers is **not allowed**.

1. Pointer Increment (`ptr++`)

Explanation

When a pointer is incremented, it moves to the **next memory location of its data type**, not the next byte.

Example

```
#include <stdio.h>


int main() {
   int a[3] = {10, 20, 30};
   int *p = a;


   printf("%d ", *p);   // 10
   p++;
   printf("%d ", *p);   // 20


   return 0;
}
```

2. Pointer Decrement (`ptr--`)

Explanation

When a pointer is decremented, it moves to the **previous memory location of its data type**.

Example

Pointer Decrement (`ptr--`)

Explanation

Pointer decrement is a pointer arithmetic operation in C that **moves the pointer to the previous memory location of the same data type**. When a pointer is decremented using the `--` operator, the address stored in the pointer is reduced by the **size of the data type** it points to. This operation is commonly used to **traverse an array in the reverse direction**.

How It Works

- If `ptr` is an `int *`, then `ptr--` moves the pointer **back by 4 bytes** (on most systems).
- If `ptr` is a `char *`, then `ptr--` moves the pointer **back by 1 byte**.
- The decrement is always based on the **data type of the pointer**, not by one byte.

Example

```
#include <stdio.h>


int main() {
   int a[3] = {10, 20, 30};
   int *p = &a[2];   // points to last element


   printf("%d ", *p);   // 30
   p--;
   printf("%d", *p);    // 20


   return 0;
}
```

Pointer Addition and Subtraction in C

Meaning

Pointer addition and subtraction are pointer arithmetic operations that allow a pointer to **move forward or backward by a specified number of elements** in a contiguous memory block such as an array. These operations are commonly used to **access array elements using pointers**. The movement of the pointer depends on the **data type** of the pointer.

---

1. Pointer Addition (`ptr + n`)

Explanation

Pointer addition moves the pointer **forward by `n` elements**. It does **not add `n` bytes** to the address; instead, it adds `n × sizeof(data_type)` bytes.

Example

```
#include <stdio.h>


int main() {
   int a[5] = {10, 20, 30, 40, 50};
   int *p = a;


   printf("%d\n", *(p + 2));  // Access 3rd element
   return 0;
} Output: 30
```

2. Pointer Subtraction (`ptr - n`)

Explanation

Pointer subtraction moves the pointer **backward by `n` elements**. Similar to addition, subtraction depends on the **size of the data type**.

Example

```
#include <stdio.h>


int main() {
```

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

    int *p = &a[4];


    printf("%d\n", *(p - 3));  // Access 2nd element

    return 0;

} Output: 20
```

Difference Between Two Pointers (`ptr1 - ptr2`)

Explanation

Subtracting one pointer from another gives the **number of elements between them**, provided both pointers point to elements of the **same array**.

Example

```
#include <stdio.h>


int main() {

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

    int *p1 = &a[4];

    int *p2 = &a[1];


    printf("%ld", p1 - p2);

    return 0;
```

} Pointers and Arrays in C

Explanation

In C programming, **pointers and arrays are closely related**. The name of an array itself acts as a **pointer to its first element**. This means that when an array is used in expressions or passed to functions, it represents the **base address** of the array. Using pointers with arrays allows efficient access and traversal of array elements and forms the basis for advanced operations such as passing arrays to functions and dynamic memory handling.

Relationship Between Pointers and Arrays

- Array name → points to the **first element** of the array
- `a[i]` is equivalent to `*(a + i)`
- Pointer arithmetic can be used to access array elements

Example: Accessing Array Using Pointer

```
#include <stdio.h>


int main() {
   int a[5] = {10, 20, 30, 40, 50};
   int *p = a;   // points to first element


   for (int i = 0; i < 5; i++) {
      printf("%d ", *(p + i));
   }


   return 0;
}
```

Output: 10 20 30 40 50

Example: Pointer Traversal of Array

```
#include <stdio.h>


int main() {
   int a[3] = {5, 15, 25};
   int *p = a;


   printf("%d ", *p);    // 5
```

```
    p++;
    printf("%d ", *p);    // 15
    p++;
    printf("%d ", *p);    // 25

    return 0;
}
```

Pointers and Strings in C

Explanation

In C programming, **strings are closely associated with pointers**. A string in C is a **character array terminated by a null character (`'\0'`)**, and the **name of the string acts as a pointer to its first character**. Using pointers with strings allows efficient string handling, easy traversal, and passing strings to functions without copying the entire string.

When a pointer is used with a string, it stores the **address of the first character** of the string. The pointer can then be incremented to access subsequent characters until the null character is encountered.

String Using Character Array

```
char str[] = "Hello";
```

- `str` is an array
- Memory is allocated for all characters including `'\0'`
- Contents can be modified

String Using Pointer

```
char *str = "Hello";
```

```
#include <stdio.h>


int main() {
```

```
    char *str = "C Language";


    while (*str != '\0') {
        printf("%c", *str);
        str++;
    }


    return 0;
}o/p C Language
```

Example: Passing String to Function Using Pointer

```
#include <stdio.h>


void display(char *s) {
    printf("%s", s);
}


int main() {
    char name[] = "Programming";
    display(name);
    return 0;
}
```

Pointer to Pointer in C

Explanation

A **pointer to pointer** is a pointer variable that **stores the address of another pointer**. In other words, it is a **double-level pointer**. While a normal pointer stores the address of a variable, a pointer to pointer stores the address of that pointer. Pointer to pointer is commonly used in **dynamic memory allocation, multidimensional arrays, and passing pointers to functions**.

Declaration

```
data_type **pointer_name;
```

ExampleVariable

x → Pointer p → Pointer pp

x &x &p

```
#include <stdio.h>


int main() {
   int x = 10;
   int *p = &x;
   int **pp = &p;


   printf("Value of x = %d\n", x);
   printf("Using p = %d\n", *p);
   printf("Using pp = %d\n", **pp);


   return 0;
}
```

Output: Value of x = 10

Using p = 10

Using pp = 10

Explanation of Example

- `x` stores value `10`
- `p` stores the address of `x`
- `pp` stores the address of pointer `p`
- `*pp` gives address of `x`
- `**pp` gives value of `x`

Applications of Pointer to Pointer

- Dynamic memory allocation (`malloc` returning pointer)
- Handling multi-dimensional arrays
- Passing pointer variables to functions
- Implementing data structures like linked lists
- **Array of Pointers in C**
- ---
- **Explanation**
- An **array of pointers** is an array in which **each element is a pointer**. Instead of storing actual data values, the array stores **addresses of variables or addresses of strings**. This concept is widely used when handling **multiple strings**, dynamic data structures, and efficient memory management.
- Each pointer in the array can point to a different memory location, and the size of the array depends on the **number of pointers**, not on the size of the data being pointed to.
- ---

Declaration

```
data_type *array_name[size];
```

Example

```
int *arr[3];
```

Example: Array of Integer Pointers

```
#include <stdio.h>


int main() {
   int a = 10, b = 20, c = 30;
   int *arr[3];

   arr[0] = &a;
   arr[1] = &b;
   arr[2] = &c;

   for (int i = 0; i < 3; i++) {
```

```
        printf("%d ", *arr[i]);
    }

    return 0;
}
```

Output: 10 20 30

Example: Array of Strings

```
#include <stdio.h>

int main() {
    char *names[3] = {"C", "Java", "Python"};

    for (int i = 0; i < 3; i++) {
        printf("%s\n", names[i]);
    }

    return 0;
}
```

Meaning of Dynamic Memory Allocation

Dynamic memory allocation in C allows programs to **request memory at runtime** instead of compile time. This is useful when the size of data is **not known in advance**. Dynamic memory is allocated from the **heap memory** using standard library functions defined in the header file **`<stdlib.h>`**.

- When memory size is not fixed
- Efficient use of memory
- Handling large data dynamically
- Creating dynamic data structures (linked lists, stacks)

1. `malloc()` – Memory Allocation

Explanation

`malloc()` (memory allocation) allocates a **single block of memory** of the specified size in bytes. The allocated memory contains **garbage values** and must be type-cast before use.

Syntax

```
ptr = (data_type *)malloc(size_in_bytes);
```

Example

```
#include <stdio.h>

#include <stdlib.h>


int main() {

   int *p;

   p = (int *)malloc(5 * sizeof(int));


   if (p == NULL) {

      printf("Memory not allocated");

      return 1;

   }


   for (int i = 0; i < 5; i++) {

      p[i] = i + 1;

      printf("%d ", p[i]);

   }


   free(p);

   return 0;

}
```

`calloc()` – Contiguous Allocation

Explanation

`calloc()` allocates memory for **multiple elements** and **initializes all bytes to zero**. It is mainly used when default zero initialization is required.

Syntax

```
ptr = (data_type *)calloc(n, size_of_each_element);
```

Example

```
#include <stdio.h>

#include <stdlib.h>


int main() {

   int *p;

   p = (int *)calloc(5, sizeof(int));


   for (int i = 0; i < 5; i++) {

      printf("%d ", p[i]);  // all zeros

   }


   free(p);

   return 0;

}
```

3. `free()` – Memory Deallocation

Explanation

`free()` is used to **release dynamically allocated memory** back to the heap. It helps prevent **memory leaks**.

Syntax

free(ptr);

Example

```
free(p);
```

p = NULL;

Aspect	`malloc()`	`calloc()`
Full form	Memory Allocation	Contiguous Allocation
Number of arguments	1	2
Syntax	`malloc(size_in_bytes)`	`calloc(n, size)`
Memory allocation	Single block	Multiple contiguous blocks
Initialization	Garbage values	All elements initialized to zero
Speed	Faster	Slightly slower
Type casting	Required in C++	Required in C++
Memory size	Total bytes specified	Calculated as `n × size`
Use case	When initial values not needed	When zero initialization needed
Return value	Pointer to allocated memory	Pointer to allocated memory
Header file	`<stdlib.h>`	`<stdlib.h>`

UNIT-IV

PART – A: User-Defined Data Types

1. Introduction to User-Defined Data Types

User-defined data types in C allow programmers to create new data types according to program requirements. They help in organizing related data, improving program readability, and reducing complexity.

Examples of user-defined data types:

- Structure
- Union
- Enumeration (enum)
- typedef (conceptual)

Structure (Paragraph with Example)

A **structure** in C is a user-defined data type that allows grouping of variables of **different data types** under a single name. It is mainly used to represent a record where all the members are logically related. For example, to store student information such as roll number, name, and marks, using separate variables becomes difficult to manage. By using a structure, all

these related data items can be combined into a single unit, making the program more organized, readable, and easy to maintain.

Syntax of Structure

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

```
struct Student {
   int roll;
   char name[20];
   float marks;
};
```

```
struct Student s1 = {101, "Ravi", 85.5};
```

In the above example, `Student` is a structure that contains three members: `roll`, `name`, and `marks`. The variable `s1` stores all the details of a student in one place, which simplifies data handling and is widely used in real-world applications like student records and employee databases.

Creating Structure Variables (Paragraph with Example)

After declaring a structure, variables of that structure type can be created to store actual data. These variables represent individual records of the structure. Each structure variable can hold values for all the members defined inside the structure. Creating structure variables allows the programmer to work with grouped data as a single unit, which improves clarity and efficiency in programs.

Example:

```
struct Student {
    int roll;
    char name[20];
```

```
    float marks;
};

struct Student s1, s2;
```

In this example, `s1` and `s2` are structure variables of type `Student`. Each variable can store separate details of different students, such as roll number, name, and marks.

Initialization of Structure (Paragraph with Example)

Initialization of a structure means assigning values to its members at the time of declaring the structure variable. This allows the structure variable to start with predefined values, avoiding the need to assign values individually later in the program. Structure initialization makes the code concise, clear, and less error-prone, especially when all required data is already available.

Example:

```
struct Student {
    int roll;
    char name[20];
    float marks;
};

struct Student s1 = {101, "Ravi", 85.5};
```

In this example, the structure variable `s1` is initialized with values for all its members in the same order as they are declared inside the structure. The roll number is set to `101`, the name to `"Ravi"`, and the marks to `85.5`.

Accessing Structure Members (Paragraph with Example)

Accessing structure members means retrieving or modifying the values stored in the individual members of a structure variable. In C, structure members are accessed using the **dot (.) operator**, which connects the structure variable name with its member name. This allows each member of the structure to be used just like a normal variable within the program.

```
struct Student {
    int roll;
    char name[20];
    float marks;
};

struct Student s1 = {101, "Ravi", 85.5};

printf("%d\n", s1.roll);
printf("%s\n", s1.name);
printf("%f\n", s1.marks);
```

In this example, the dot operator is used to access the `roll`, `name`, and `marks` members of the structure variable `s1`. These values can be displayed, modified, or used in calculations as required.

Array of Structures (Paragraph with Example)

An **array of structures** is a collection of structure variables of the same type stored under a single array name. It is used when multiple records of the same kind need to be stored and processed, such as details of many students or employees. Each element of the array represents one complete structure, and its members can be accessed using the array index along with the dot operator.

Example:

```
struct Student {
    int roll;
    char name[20];
    float marks;
};
```

```
struct Student s[3];


s[0].roll = 101;

s[1].roll = 102;
```

In this example, `s` is an array of structures that can store details of three students. Each student's data is accessed using an index like `s[0]`, `s[1]`, etc., followed by the dot operator to refer to individual members.

Advantages of Structures

- Structures allow grouping of related data items of different data types under a single name.
- They improve program readability by organizing complex data in a clear and meaningful way.
- Structures make data handling easier by treating multiple related variables as one unit.
- They support real-world data representation such as student records, employee details, and product information.
- Structures can be used with arrays and functions, enabling efficient processing of large amounts of data.
- They help in reducing program complexity and improving maintainability.

Unions (Paragraph with Example)

A **union** is a user-defined data type in C that allows different data types to share the **same memory location**. Unlike structures, where each member has its own memory, all members of a union use a single shared memory space. As a result, at any given time, only **one member** of the union can hold a valid value. Unions are mainly used when memory optimization is important, especially in systems programming and embedded applications.

Example:

```
union Data {

    int i;

    float f;

    char c;
```

};

union Data d;

d.i = 10;

In this example, the union `Data` can store an integer, a float, or a character, but only one value at a time. When `d.i` is assigned a value, the same memory location is used for all members, and the previously stored value (if any) is overwritten.

Declaration of Union Variables (Paragraph with Example)

After declaring a union, variables of that union type can be created to store data. Declaring union variables allows the program to use the union and assign values to its members. Since all members of a union share the same memory location, only one member should be used at a time through the union variable.

Example:

union Data {

int i;

float f;

char c;

};

union Data d1, d2;

In this example, `d1` and `d2` are union variables of type `Data`. Each variable can store one value at a time, either an integer, a float, or a character, using the shared memory space of the union.

Accessing Union Members (Paragraph with Example)

Accessing union members means reading or assigning values to the individual members of a union variable. In C, union members are accessed using the **dot (.) operator**, similar to structures. However, since all members of a union share the same memory location, only one

member should be accessed at a time; accessing another member may overwrite the previous value.

Example:

```
union Data {
    int i;
    float f;
    char c;
};

union Data d;

d.i = 25;
printf("%d\n", d.i);

d.f = 10.5;
printf("%f\n", d.f);
```

In this example, the union variable `d` first stores an integer value using `d.i`. When a float value is assigned using `d.f`, it overwrites the previous integer value because both members share the same memory.

Comparison Table: Structure vs Union (Detailed)

Basis of Comparison	Structure	Union
Type	User-defined data type	User-defined data type
Memory allocation	Each member gets separate memory	All members share the same memory
Total size	Equal to the sum of sizes of all members	Equal to the size of the largest member
Data storage	All members can hold values at the same time	Only one member can hold a value at a time
Data access	Members can be accessed	Accessing one member affects

	independently	others
Memory efficiency	Less memory efficient	Highly memory efficient
Data safety	High, no data loss	Low, data may be overwritten
Use in real-world applications	Used to represent records like student, employee	Used in memory-critical applications
Complexity	Easy to understand and use	Slightly complex due to shared memory
Performance	Slower compared to union (more memory)	Faster access due to less memory usage
Modification impact	Changing one member does not affect others	Changing one member overwrites others
Typical applications	Databases, management systems	Embedded systems, compilers
Example	Student details	Variant data storage

Enumeration (enum)

An **enumeration (enum)** is a user-defined data type in C used to define a collection of **named integer constants**. It allows a variable to take one value from a predefined set of meaningful names instead of arbitrary numbers. Enumerations improve **code clarity and readability**, as the names used clearly represent their purpose. They also help in **error reduction**, because only valid predefined values are assigned to enum variables. Internally, enum constants are stored as integers, making them efficient in terms of memory and performance.

By default, the first enumerator is assigned the value `0`, and the subsequent enumerators are assigned increasing integer values. However, programmers can also explicitly assign values to enum constants when required.

Example:

```
#include <stdio.h>


enum Signal {RED, YELLOW, GREEN};


int main() {
    enum Signal light;
```

```
    light = GREEN;


    if (light == RED)
        printf("Stop the vehicle");
    else if (light == YELLOW)
        printf("Get ready");
    else if (light == GREEN)
        printf("Go");


    return 0;
}
```

Explanation:

In this example, `Signal` is an enumeration that defines three named constants: `RED`, `YELLOW`, and `GREEN`. By default, `RED` is assigned `0`, `YELLOW` is `1`, and `GREEN` is `2`. The variable `light` can store only one of these values. When `light` is set to `GREEN`, the corresponding message **"Go"** is displayed. This approach makes the program easy to read, avoids invalid values, and clearly represents real-world conditions.

typedef –

typedef is a keyword in C used to create an **alias (alternative name)** for an existing data type. It does not create a new data type; instead, it provides a more meaningful or shorter name for an existing one. The main purpose of `typedef` is to improve **code readability, portability, and ease of use**, especially when dealing with complex data types like structures, unions, pointers, and arrays.

Syntax

```
typedef existing_data_type new_name;
```

typedef with Structure (Good Example)

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

```
typedef struct {
    int id;
    char name[20];
    float salary;
} Employee;


int main() {
    Employee e1 = {101, "Ravi", 25000.50};


    printf("ID: %d\n", e1.id);
    printf("Name: %s\n", e1.name);
    printf("Salary: %.2f\n", e1.salary);


    return 0;
}
```

Explanation:

In this example, `typedef` is used to give the structure a new name `Employee`. Because of `typedef`, we can directly declare structure variables using `Employee` instead of writing `struct Employee` every time. This makes the program **cleaner and easier to understand**, especially in large projects.

Advantages of typedef

- Improves readability of the code
- Simplifies complex declarations
- Makes code portable across systems
- Reduces repetition of long data type names

File Handling in C

File handling in C refers to the process of **creating, opening, reading, writing, and closing files** stored on secondary storage devices such as hard disks or SSDs. Unlike variables, which

store data temporarily in main memory, files allow **permanent storage of data**, even after the program terminates. File handling is essential for applications like record management systems, billing software, and data logging programs.

1. Types of Files in C

C mainly supports two types of files:

a) Text Files

- Store data in human-readable format
- Data is stored as characters (ASCII codes)
- Example: `.txt` files

b) Binary Files

- Store data in binary (machine-readable) format
- Faster and more secure
- Example: `.dat`, `.bin` files

2. File Pointer

A **file pointer** is a pointer of type `FILE` that is used to **connect a C program with a file**. It acts as a link between the program and the file stored on secondary memory. The file pointer keeps track of important information such as the current position in the file, the file mode, and the status of the file during file operations like reading and writing.

In C, file pointers are defined using the `FILE` structure provided in the header file `stdio.h`. Without a file pointer, a program cannot perform any file operations.

Syntax:

```
FILE *fp;
```

Example:

```
FILE *fp;

fp = fopen("sample.txt", "r");
```

Explanation:
In this example, `fp` is a file pointer that points to the file `sample.txt`. When the file is opened using `fopen()`, the file pointer stores the address of the file and allows the program to read data from it. All file operations such as reading, writing, and closing the file are performed using this file pointer.

3. Opening a File

Opening a file is the first step in file handling. In C, a file is opened using the `fopen()` function, which establishes a connection between the program and the file through a file pointer. The `fopen()` function can be used either to **open an existing file** or to **create a new file**, depending on the mode specified. If the file cannot be opened, the function returns `NULL`.

Syntax:
file_pointer = fopen("filename", "mode");

Example:

FILE *fp;

fp = fopen("data.txt", "w");

Explanation:
In this example, the file `data.txt` is opened in **write mode (`"w"`)**. If the file already exists, its contents will be erased; if it does not exist, a new file will be created. The file pointer `fp` stores the reference to the opened file and is used for further file operations such as writing or reading data.

4. Closing a File

After file operations, the file must be closed using `fclose()`.

Syntax: fclose(fp);

5. Writing Data to a File

Writing data to a file means storing information permanently on secondary storage. In C, data can be written to a file using different functions depending on whether the file is a **text file** or a **binary file**. Writing data is possible only when the file is opened in a suitable mode such as **write (`"w"`)** or **append (`"a"`)** mode. The data written to the file remains stored even after the program terminates.

Example (Writing to a Text File):

```
#include <stdio.h>
int main() {
    FILE *fp;
    fp = fopen("message.txt", "w");
    fprintf(fp, "Welcome to File Handling in C");
    fclose(fp);

    return 0;
}
```

Explanation:
In this example, the file message.txt is opened in write mode. The fprintf() function is used to write text into the file. After writing the data, the file is closed using fclose(), which ensures that the data is saved permanently on the storage device.

File Input and Output Operations in C

File input and output (I/O) operations in C are used to read data from files and write data into files stored on secondary storage. These operations allow programs to handle large amounts of data permanently. File I/O is performed using a **file pointer (`FILE *`)** and a set of predefined library functions available in the `stdio.h` header file.

1. File Input Operations (Reading Data from a File)

File input operations are used to read data from an existing file. The file must be opened in **read (`"r"`)** or **read-related modes**.

Common File Input Functions

- `fscanf()` – Reads formatted data from a text file
- `fgets()` – Reads a line of text from a file
- `fread()` – Reads data from a binary file

Example (File Input):

```
#include <stdio.h>
int main() {
   FILE *fp;
   char str[50];
   fp = fopen("message.txt", "r");
   fgets(str, 50, fp);
   printf("%s", str);
   fclose(fp);

   return 0;
}
```

2. File Output Operations (Writing Data to a File)

File output operations are used to write data into a file. The file must be opened in **write (`"w"`)** or **append (`"a"`)** mode.

Common File Output Functions

- `fprintf()` – Writes formatted data to a text file
- `fputs()` – Writes a string to a file
- `fwrite()` – Writes data to a binary file

Example (File Output):

```
FILE *fp;
fp = fopen("output.txt", "w");
fprintf(fp, "File Input and Output Operations");
fclose(fp);
```

Binary File I/O Example (with Explanation)

Binary file input/output is used to read and write data in **binary (machine-readable) format**. It is faster and more efficient than text file I/O and is commonly used to store **structures, numbers, and large data blocks**. In C, binary file operations are performed using the functions `fwrite()` and `fread()`.

Example: Writing and Reading Data from a Binary File

```
#include <stdio.h>

struct Student {
    int roll;
    float marks;
};

int main() {
    FILE *fp;
    struct Student s1 = {101, 85.5}, s2;

    /* Writing to binary file */
    fp = fopen("student.dat", "wb");
    fwrite(&s1, sizeof(s1), 1, fp);
    fclose(fp);

    /* Reading from binary file */
    fp = fopen("student.dat", "rb");
    fread(&s2, sizeof(s2), 1, fp);
    fclose(fp);
    printf("Roll: %d\n", s2.roll);
    printf("Marks: %.2f\n", s2.marks);

    return 0;
}
```

Explanation:

1. A structure `Student` is defined with `roll` and `marks`.
2. The file `student.dat` is opened in **write binary mode (`"wb"`)**.
3. The `fwrite()` function writes the entire structure `s1` into the binary file.

4. The file is closed to save data properly.
5. The file is reopened in **read binary mode (`"rb"`)**.
6. The `fread()` function reads the data from the file into structure `s2`.
7. The stored values are displayed on the screen.