INDEX

1.1 Introduction to Programming

Programming is the process of designing and writing a set of instructions that tell a computer how to perform a particular task. These instructions are written using a programming language that the computer can understand and execute. A program is a collection of such instructions arranged in a logical sequence to solve a specific problem.

Computers cannot understand human languages directly. Therefore, programmers use programming languages to communicate with the computer. A programming language provides a set of rules, symbols, and syntax that allow programmers to write instructions in a structured and organized manner.

Programming plays a very important role in modern computing. It is used to develop software applications, operating systems, websites, mobile applications, and many other technological systems. Through programming, complex problems can be solved efficiently by breaking them into smaller logical steps.

The main objective of programming is to convert a problem into a series of clear instructions so that a computer can process the data and produce the required output. A good program should be easy to understand, efficient, reliable, and easy to maintain.

Programming involves several steps such as problem analysis, algorithm design, coding, testing, debugging, and maintenance. By following these steps, programmers can develop effective and error-free programs.

In simple terms, programming is the method of instructing a computer to perform useful tasks by writing programs using programming languages.

1.1.1 Types of Programming Languages

Programming languages are used to communicate instructions to a computer so that it can perform specific tasks. Over time, different types of programming languages have been developed to make programming easier, faster, and more efficient. These languages are generally classified based on their level of abstraction and the way they interact with computer hardware.

1. Based on Level of Language

1. Machine Language

Machine language is the lowest level programming language and is directly understood by the computer's processor. It consists of binary digits (0s and 1s) that represent instructions and data. Each instruction in machine language corresponds to a specific operation that the computer can perform. Since it is executed directly by the computer, machine language is very fast. However, it is difficult for humans to write and understand because it requires remembering complex binary codes.

2. Assembly Language

Assembly language is a low-level programming language that uses symbolic codes instead of binary numbers. These symbolic instructions are easier to understand than machine language. An assembler is required to convert assembly language programs into machine language before they can be executed by the computer. Although assembly language is easier than machine language, it is still closely related to hardware and requires detailed knowledge of computer architecture.

3. High-Level Language

High-level languages are designed to be more user-friendly and closer to human language. They use simple English-like words and mathematical symbols, which makes programming easier to learn and write. High-level languages are machine independent, meaning the same program can run on different types of computers with minimal changes. Examples of high-level languages include C, C++, Java, and Python. A compiler or interpreter is used to translate high-level language programs into machine language so that the computer can execute them.

2. Based on Programming Paradigm

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*

1.1.2 History of C Language

*The C programming language is one of the most popular and powerful programming languages used in computer science. It was developed in the early 1970s at **Bell Laboratories (Bell Labs)** in the United States. The language was created by **Dennis Ritchie** in **1972**.*

*The development of C language is connected with earlier programming languages such as **BCPL (Basic Combined Programming Language)** and **B language**. In 1967, BCPL was developed by **Martin Richards** for system programming. Later, in 1970, **Ken Thompson** created the B language at Bell Labs to develop the early versions of the **UNIX operating system**. However, the B language had several limitations and lacked important features.*

*To improve the B language, Dennis Ritchie designed the C programming language. C included new features such as **data types, structured programming, and better memory management**, which made programming easier and more efficient.*

*One of the major uses of C language was in developing the **UNIX operating system**. Earlier, operating systems were usually written in assembly language. When UNIX was rewritten using C, it became easier to modify and run on different types of computers. This made C very popular among programmers.*

*As the popularity of C increased, it became necessary to create a standard version of the language. In **1989**, the **American National Standards Institute (ANSI)** introduced a standard version called **ANSI C**. Later, the **International Organization for Standardization (ISO)** also adopted this standard.*

*Today, C is widely used in developing **operating systems, system software, embedded systems, and compilers**. Many modern programming languages such as **C++, Java, and C#** are influenced by the concepts of C language. Because of its simplicity and efficiency, C is still considered one of the most important programming languages for learning programming fundamentals.*

1.1.3 Structure of a C Program

*The **structure of a C program** refers to the standard arrangement of different sections that together form a complete program. This structured format helps programmers organize their code in a logical manner and allows the compiler to interpret the program correctly. A well-structured program improves **clarity, readability, debugging, and maintenance**.*

*C is known as a **structured programming language**, which means programs are written in a systematic format using functions and well-defined sections. Each section of a C program performs a specific task during program development and execution. Although all sections may not always appear in every program, understanding them helps programmers design efficient and well-organized programs.*

A typical structure of a C program consists of the following sections:

1. *Documentation Section*
2. *Link Section*
3. *Definition Section*
4. *Global Declaration Section*
5. *main() Function Section*
6. *User-Defined Function Section*

General Structure of a C Program

```
/* Documentation Section */

#include <stdio.h>
#define MAX 100

/* Global Declaration Section */
int total;
void display();

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

/* User Defined Function */
void display()
{
   printf("Hello C Programming");
}
```

1. Documentation Section

*The **documentation section** appears at the beginning of the program and contains comments that describe the program. This section is used to provide information about the program such as the **program name, purpose, author's name, date, and description of the program logic**.*

Comments are written using / */ for multi-line comments or // for single-line comments. These comments are ignored by the compiler and do not affect the execution of the program.*

The main objective of the documentation section is to make the program easier to understand for programmers who read or modify the code later. Good documentation is considered an important practice in software development because it helps in maintaining large programs.

Example:

```
/* Program to display a message in C language */
```

2. Link Section

*The **link section** is used to include header files required by the program. Header files contain declarations of **library functions, macros, and constants** that are provided by the C standard library.*

The #include directive is used to include header files in a program. During compilation, the compiler connects the program with the required library files so that the functions defined in those libraries can be used.

*For example, the header file **stdio.h** contains declarations for standard input and output functions such as printf() and scanf().*

Example:

#include <stdio.h>

If the required header file is not included, the compiler may not recognize the library functions used in the program.

3. Definition Section

*The **definition section** is used to define symbolic constants using the #define directive. A symbolic constant is a name that represents a fixed value used throughout the program.*

Using symbolic constants improves program readability and makes it easier to modify values when required. If a constant value needs to be changed, it can be updated in one place instead of modifying multiple lines of code.

Example:

#define MAX 100

*In this example, MAX represents the value **100** and can be used anywhere in the program.*

*This section is also handled by the **C preprocessor**, which processes these directives before the compilation stage.*

4. Global Declaration Section

*The **global declaration section** contains declarations of global variables and function prototypes that can be accessed by all functions in the program.*

*A **global variable** is a variable declared outside all functions. Such variables remain available throughout the program and can be accessed by multiple functions.*

*Function prototypes are also written in this section. A function prototype informs the compiler about the **function name, return type, and parameters** before the function is actually defined.*

Example:

int total;

void display();

Here:

- *total is a global variable.*
- *display() is a function prototype.*

Global declarations are useful when several functions need to share common data.

5. main() Function Section

*The **main() function** is the most important part of every C program because it serves as the **starting point of program execution**. When a program runs, the compiler first executes the statements inside the main() function.*

*Every C program must contain **exactly one main() function**. The main() function can include:*

- *Variable declarations*
- *Input and output operations*
- *Program logic and calculations*
- *Function calls*

Example:

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

The statement return 0; indicates that the program has terminated successfully and returns control to the operating system.

6. User-Defined Function Section

User-defined functions are functions written by the programmer to perform specific tasks within the program. These functions help divide a large program into smaller parts, making the program easier to understand and manage.

*Functions improve **modularity and reusability** of code. Instead of writing the same code multiple times, a function can be called whenever needed.*

Example:

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

In this example, the function display() prints a message when it is called from the main() function.

Using user-defined functions offers several advantages:

- *Reduces code repetition*
- *Improves program organization*
- *Simplifies debugging*
- *Makes the program easier to maintain*

1.1.4 Programming Rules

Programming rules are the basic principles and guidelines that must be followed while writing programs in the C language. These rules ensure that programs are written in a proper format so that the compiler can understand the instructions and execute them correctly. By following these rules, programmers can develop programs that are ***structured, readable, efficient, and easy to maintain****.*

C is a ***structured and procedural programming language****, which means that programs are written in a logical sequence of statements and functions. The compiler checks the syntax of each statement, and if the rules are not followed correctly, the program will produce errors during compilation. Therefore, understanding and applying programming rules is an essential part of learning C programming.*

The important programming rules of C are explained below.

1. Every C Program Must Contain a main() Function

The main() function is the starting point of execution in every C program. When a program is executed, the operating system transfers control to the main() function, and all statements inside this function are executed sequentially.

A C program cannot run without the main() function because it acts as the ***entry point*** *of the program. The main() function may contain variable declarations, program logic, input/output operations, and function calls.*

Example:

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

The statement return 0; indicates that the program has executed successfully and returns control to the operating system.

2. Each Statement Must End with a Semicolon

In C programming, a semicolon (;) is used to indicate the end of a statement. The compiler reads each statement until it encounters a semicolon. If the semicolon is missing, the compiler cannot recognize the end of the statement and produces a syntax error.

Example:

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

Here, each statement ends with a semicolon, which marks the completion of the instruction.

3. C Language is Case Sensitive

C programming language treats uppercase and lowercase letters as different characters. This means that identifiers written with different cases are treated as separate variables or functions.

Example:

int value;
int Value;

In this example, value and Value are considered two different variables.

Because of case sensitivity, programmers must be careful while writing variable names, function names, and keywords.

4. Variables Must Be Declared Before They Are Used

Before a variable can be used in a C program, it must be declared with an appropriate data type. The declaration informs the compiler about the variable name and the type of data it will store.

Example:

int number;
number = 25;

If a variable is used without declaring it first, the compiler will generate an error because it does not know the type of data associated with that variable.

5. Identifiers Must Follow Naming Rules

Identifiers are names given to program elements such as variables, functions, arrays, and structures. C language defines certain rules for naming identifiers.

The important rules are:

- *An identifier must begin with a **letter (A–Z or a–z)** or an **underscore (_)**.*
- *It cannot start with a **number**.*

- *It cannot contain **spaces or special characters**.*
- *It must not use **reserved keywords** of the C language.*
- *Identifiers should be meaningful and descriptive.*

Examples of valid identifiers:

total
count1
student_name

Examples of invalid identifiers:

1number
total-value
int

6. Use of Comments in Programs

Comments are used to explain the purpose of statements or sections of a program. They help programmers understand the logic of the program easily.

Comments are ignored by the compiler and therefore do not affect program execution. In C language, comments can be written in two ways:

Single-line comment

// This is a comment

Multi-line comment

/ This program calculates the sum of two numbers */*

Comments are especially useful when programs become large and complex.

7. Use Proper Indentation and Formatting

Indentation refers to arranging program statements in a structured and visually clear manner using spaces or tabs. Although indentation does not affect the execution of a program, it greatly improves readability.

Proper indentation helps programmers understand the structure of loops, conditional statements, and functions.

Example:

```
int main()
{
   int a = 5;
   int b = 10;

   printf("Sum = %d", a + b);

   return 0;
}
```

Well-formatted programs are easier to debug and maintain.

8. Follow Structured Programming Principles

C language supports structured programming, which means programs should be organized into ***functions and logical blocks****. Instead of writing the entire program in a single block, the program should be divided into smaller modules or functions.*

This approach provides several advantages:

- *Reduces program complexity*
- *Improves readability*
- *Makes debugging easier*
- *Encourages code reuse*

Using functions allows programmers to develop large programs more efficiently.

9. Use Meaningful Variable Names

Meaningful variable names improve program clarity and make it easier for programmers to understand the program logic.

For example:

```
int totalMarks;
int studentAge;
```

These names clearly describe the purpose of the variables, making the program easier to read.

10. Avoid Unnecessary Complexity in Programs

Programs should be written in a simple and logical manner. Avoid writing overly complicated statements when simpler solutions are possible. Simple programs are easier to understand, debug, and maintain.

1.2 Program Development Tools

1.2.1 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.

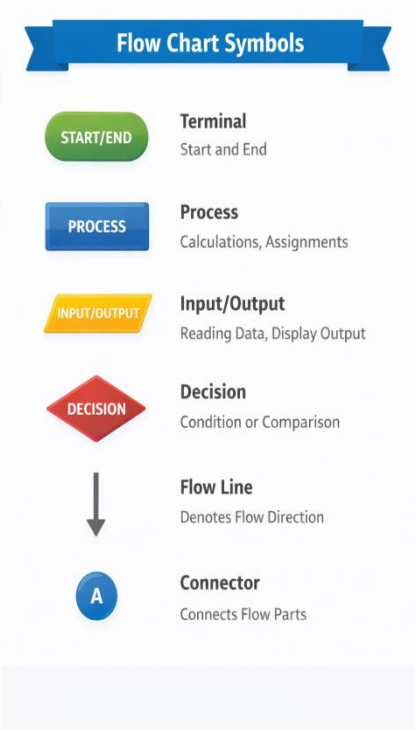
Flow Chart Symbols

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.

1.2.2 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

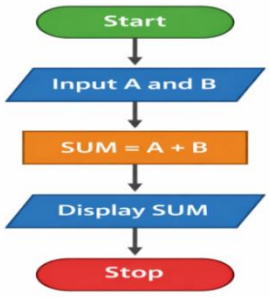
Step 1: *Start*
Step 2: *Read two numbers A and B*
Step 3: *Calculate SUM = A + B*
Step 4: *Display SUM*
Step 5: *Stop*

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.

1.3 Steps Involved in Program Execution

1.3 Executing a C Program

*Executing a C program refers to the complete process through which a program written by a programmer in the C language is transformed into machine-level instructions and then executed by the computer to produce the desired output. Since C is a **high-level programming language**, the instructions written by the programmer cannot be directly understood by the computer hardware. Computers can only understand **machine language**, which consists of binary digits (0s and 1s). Therefore, the C program must be translated into machine language before it can be executed.*

*The execution of a C program involves several stages that are performed by different system software components such as the **preprocessor, compiler, linker, loader, and operating system**. Each stage performs a specific task in converting the source program into an executable program.*

The general flow of executing a C program is shown below.

Source Program (.c file)

↓

Preprocessing

↓

Compilation

↓

Object Code

↓

Linking

↓

Executable File

↓

Loading

↓

Execution

1. Source Program

*The first step in executing a C program is writing the program using a **text editor or an Integrated Development Environment (IDE)**. The programmer writes instructions according to the syntax and rules of the C programming language.*

*The program written by the programmer is called the **source program** or **source code**. It is usually saved with the extension **.c**.*

Example:

program.c

At this stage, the program is written in a form that is understandable to humans but cannot be executed directly by the computer because the computer does not understand high-level languages.
The source program contains elements such as:

- *Variables*
- *Functions*
- *Statements*
- *Control structures*
- *Input and output instructions*

These elements define the logic and functionality of the program.

2. Preprocessing

*Before the compilation process begins, the source program is processed by a program called the **C preprocessor**. The preprocessor handles special instructions known as **preprocessor directives**, which start with the symbol #.*

The main purpose of preprocessing is to prepare the program for compilation by performing several operations such as:

Header File Inclusion

The #include directive is used to include header files that contain declarations of standard library functions.

Example:

#include <stdio.h>

This directive allows the program to use functions such as printf() and scanf().

Macro Definition

The #define directive is used to define symbolic constants or macros.

Example:

#define MAX 100

*During preprocessing, all occurrences of MAX in the program are replaced with the value **100**.*

Comment Removal
All comments written in the program are removed during preprocessing because they are not required for compilation.
Example:

/ This is a comment */*

Conditional Compilation
The preprocessor also allows selective compilation using directives such as:
#ifdef
#ifndef
#endif

After preprocessing is completed, the modified source code is passed to the compiler.

3. Compilation

*The **compiler** is responsible for translating the preprocessed C program into **object code**. Object code is a machine-level representation of the program that can be processed by the computer.*

During compilation, the compiler performs several important tasks:

- *Checking the program for **syntax errors***
- *Verifying data types*
- *Checking variable declarations*
- *Ensuring correct usage of functions and operators*
- *Translating high-level instructions into machine instructions*

If the compiler detects errors, it displays error messages that indicate the location and type of error. The programmer must correct these errors before the program can be successfully compiled.

*After successful compilation, the compiler generates an **object file**. However, this object file is not yet a complete executable program because it may require external library functions.*

4. Linking

*The next stage is **linking**, which is performed by a system program called the **linker**.*

*The linker combines the **object code generated by the compiler** with the required **library files**. Many programs use predefined functions such as printf() and scanf(), which are stored in standard libraries.*

The linker performs the following tasks:

- *Connecting object code with library functions*
- *Resolving references to external functions and variables*
- *Combining different object modules*

*After successful linking, the linker generates a **final executable file**.*
Example:

program.exe

This executable file contains machine instructions that can be executed by the computer.

5. Loading

*After the executable file is created, it must be transferred into the computer's **main memory (RAM)** so that the processor can access it. This process is known as **loading**.*

*Loading is performed by the **loader**, which is a part of the operating system.*

The loader performs the following tasks:

- *Transfers the executable program into memory*
- *Allocates memory space for program instructions and variables*
- *Prepares the program for execution*

Once loading is complete, the program is ready to run.

6. Execution

Execution is the final stage of the program execution process. In this stage, the ***Central Processing Unit (CPU)*** *begins executing the instructions stored in memory.*

Execution starts from the ***main() function****, which is the entry point of every C program. The processor executes each instruction sequentially according to the program logic.*

During execution, the program may perform several operations such as:

- *Accepting input from the user*
- *Processing data*
- *Performing calculations*
- *Displaying output on the screen*

Example Output:

Hello C

When the program finishes executing all instructions, it terminates and control is returned to the operating system.

1.3.3 Preprocessors

The ***preprocessor*** *is a program that processes the C source code before the actual compilation begins. It acts as the* ***first stage in the execution of a C program****. The main role of the preprocessor is to examine the program for special instructions called* ***preprocessor directives*** *and perform certain modifications on the source code.*

Preprocessor directives are instructions that begin with the symbol #. These directives give instructions to the compiler about how the source code should be handled before compilation. Unlike normal C statements, preprocessor directives do not end with a semicolon because they are processed before the compilation stage.

Conceptually, the preprocessor acts like a ***preparation stage*** *that modifies and organizes the program so that the compiler can easily translate it into machine code. It simplifies programming by allowing programmers to include library files, define constants, and control which parts of the program should be compiled.*

The preprocessor does not execute the program; instead, it ***modifies the source code and produces an expanded version of the program****, which is then sent to the compiler for further processing.*

The position of the preprocessor in the program execution process is shown below:

Source Program

↓

Preprocessor

↓

Compiler

↓

Object Code

↓

Linker

↓

Executable Program

Major Functions of the C Preprocessor

The C preprocessor performs several important tasks before compilation.

1. File Inclusion

File inclusion allows the program to include external files that contain declarations of library functions. This is done using the ***#include directive****.*

Header files contain definitions of standard functions that can be used in programs. By including these files, programmers can use predefined functions without writing them again.

Example:

#include <stdio.h>

Here, the header file ***stdio.h*** *provides input and output functions such as printf() and scanf().*

2. Macro Definition

The preprocessor allows programmers to define symbolic names for constants or expressions using the ***#define directive****.*

Example:

#define PI 3.14

In this example, ***PI*** *represents the constant value* ***3.14****. Whenever the macro name appears in the program, the preprocessor replaces it with the defined value.*

This feature improves program readability and makes it easier to update values.

3. Macro Expansion

Macro expansion refers to replacing macro names with their corresponding values throughout the program. This replacement is done automatically during preprocessing.

Example:

*#define SQUARE(x) x*x*

*If the statement SQUARE(4) appears in the program, the preprocessor replaces it with 4*4.*

This reduces repetitive coding and simplifies program development.

4. Conditional Compilation

Conditional compilation allows certain parts of the program to be compiled only when specific conditions are satisfied. This is useful when the same program needs to run in different environments.

Some common directives used for conditional compilation are:

#ifdef
#ifndef
#if
#else
#endif

These directives help programmers control which parts of the code should be included during compilation.

5. Comment Removal

Comments written in the program are meant only for programmers to explain the code. They are not required for program execution.

*During preprocessing, the preprocessor automatically **removes all comments** from the program before sending it to the compiler.*

Example:

/ This program prints a message */*

1.4 Basic Elements of C

*The **basic elements of C** are the fundamental components that form the foundation of every C program. These elements define how instructions are written, how data is stored, and how*

operations are performed within a program. Understanding these elements is essential because they provide the basic structure required to design and develop programs in the C programming language.

In C programming, a program is composed of several building blocks such as ***keywords, identifiers, constants, and variables****. These elements help the programmer define data, perform operations, and control the flow of execution. Each element has a specific purpose and follows certain rules defined by the syntax of the C language.*

The proper use of these elements ensures that programs are written in a clear, organized, and efficient manner. These elements also help the compiler understand the meaning of the program and translate it into machine language.

The major basic elements of C include:

- *Keywords*
- *Identifiers*
- *Constants*
- *Variables*
- *Rules for defining variables*
- *Scope and lifetime of variables*

Each of these elements plays an important role in program development and execution.

1.4.1 Keywords and Identifiers

Keywords and identifiers are fundamental components of the C programming language. They are used to construct program statements and to represent different elements within a program. Keywords define the basic structure and operations of the C language, while identifiers allow programmers to assign meaningful names to program elements such as variables, functions, arrays, and structures.

Understanding keywords and identifiers is essential for writing correct and well-structured programs because they form the basic vocabulary used in C programming.

Keywords

Keywords are ***reserved words*** *in the C programming language that have predefined meanings. These words are part of the language syntax and are used by the compiler to perform specific functions such as declaring data types, controlling the flow of execution, and defining program structure.*

Since keywords are reserved by the language, they ***cannot be used as identifiers*** *for naming variables, functions, or other program elements. If a programmer attempts to use a keyword as an identifier, the compiler will generate an error.*

Keywords are recognized by the compiler during compilation and are interpreted according to their predefined roles in the language.

Examples of commonly used C keywords include:

int float char double

if else for while

do switch case break

continue returnEach keyword serves a particular purpose in program development. For instance, the keyword ***int*** *is used to declare integer variables, while* ***if*** *and* ***else*** *are used for decision-making statements in a program.*

Example:
int number = 10;
In this example, ***int*** *is a keyword that specifies the data type of the variable* ***number****.*

The C language contains a fixed set of keywords, and their meanings cannot be changed by the programmer.

Identifiers

Identifiers are ***user-defined names*** *used to identify various elements of a program. These elements may include variables, functions, arrays, structures, labels, and other program components. Identifiers allow programmers to refer to these elements easily within the program.*

Unlike keywords, identifiers are created by programmers according to the requirements of the program. The names chosen for identifiers should be meaningful so that they clearly describe the purpose of the variable or function.

Example:

int totalMarks;

In this example, ***totalMarks*** *is an identifier used as the name of a variable that stores integer values.*

Identifiers play an important role in improving the readability and clarity of a program because meaningful names help programmers understand the logic of the program easily.

Rules for Naming Identifiers

In C programming, identifiers must follow certain rules so that the compiler can correctly recognize them. These rules ensure that identifiers are valid and do not conflict with the syntax of the language.

The main rules for naming identifiers are as follows:

1. *An identifier must begin with a **letter (A–Z or a–z)** or an **underscore (_)**.*
2. *It cannot begin with a **digit**.*
3. *Identifiers may contain **letters, digits, and underscores**.*
4. *Identifiers cannot contain **spaces or special characters** such as @, #, %, etc.*
5. *Identifiers must not be **C keywords**.*
6. *Identifiers are **case-sensitive**, meaning that uppercase and lowercase letters are treated as different characters.*
7. *Identifiers should be **meaningful and descriptive** to improve program readability.*

Examples of valid identifiers:

sum
studentName
total_marks
count1

Examples of invalid identifiers:

1value
total marks
float

*In the above examples, **1value** is invalid because it begins with a digit, **total marks** is invalid because it contains a space, and **float** is invalid because it is a keyword.*

1.4.2 Constants

*In C programming, a **constant** is a value that does not change during the execution of a program. Once a constant is defined, its value remains fixed throughout the program and cannot be modified while the program is running. Constants are used to represent fixed data such as numbers, characters, or strings that are required in a program.*

Constants help improve the clarity and reliability of a program because they prevent accidental changes to important values. They also make programs easier to understand and maintain, since the meaning of the value remains consistent throughout the program.

In simple terms, a constant represents ***data whose value remains unchanged during program execution****.*

Example:

int a = 10;

In this example, ***10*** *is a constant value assigned to the variable a.*

Types of Constants in C

Type of Constant	***Description***	***Syntax***	***Example***
Integer Constants	*Whole numbers without a decimal point. They may be positive or negative.*	*int variable = integer_value;*	*int num = 10;*
Floating-Point Constants	*Numbers that contain a decimal point or fractional value.*	*float variable = decimal_value;*	*float pi = 3.14;*
Character Constants	*A single character enclosed within single quotation marks.*	*char variable = 'character';*	*char grade = 'A';*
String Constants	*A sequence of characters enclosed within double quotation marks.*	*char variable[] = "string";*	*char name[] = "C Programming";*
Enumeration Constants	*Named integer constants defined using the enum keyword.*	*enum enum_name {value1, value2,...};*	*enum day {MON, TUE, WED};*

1.4.3 Variables

In the C programming language, a ***variable*** *is a named memory location used to store data that may change during the execution of a program. Variables play an important role in*

programming because they allow programs to store values, manipulate data, and perform calculations while the program is running.

When a variable is declared, the compiler allocates a specific amount of memory in the computer's memory (RAM) to store the value of that variable. Each variable has a ***name****, a* ***data type****, and a* ***value****. The data type specifies the kind of data that the variable can store, such as integers, floating-point numbers, or characters.*

Unlike constants, which store fixed values, variables can store values that may change during program execution. For example, in a program that calculates the total marks of a student, the marks entered by the user are stored in variables so that they can be processed and updated whenever necessary.

Conceptually, a variable can be understood as a ***container used to store information temporarily in memory****. The name of the container represents the variable name, and the value stored inside the container can change whenever the program assigns a new value to it.*

Declaration of Variables

Before a variable can be used in a program, it must be declared. Variable declaration informs the compiler about the name of the variable and the type of data it will store.

General Syntax:

data_type variable_name;

Example:

int age;
float salary;
char grade;
In the above examples:

- *age is a variable used to store an integer value.*
- *salary is a variable used to store a decimal value.*
- *grade is a variable used to store a character.*

Characteristics of Variables

Variables in C have several important characteristics:

1. *Every variable must have a* ***data type*** *that defines the type of value it can store.*
2. *Each variable must have a* ***unique name*** *within its scope.*
3. *The value stored in a variable can* ***change during program execution****.*
4. *Variables occupy* ***memory space*** *in the computer.*
5. *Variable names must follow the* ***rules of identifiers****.*

1.4.4 Rules for Defining Variables

When defining variables in C, certain rules must be followed so that the compiler can correctly interpret the program.

Some important rules for defining variables include:

1. *A variable name must begin with a **letter (A–Z or a–z)** or an **underscore (_)**.*
2. *A variable name cannot begin with a **number**.*
3. *Variable names cannot contain **spaces or special characters**.*
4. *Keywords of the C language cannot be used as variable names.*
5. *Variable names should be meaningful and descriptive.*
6. *Variable names are **case-sensitive**, meaning uppercase and lowercase letters are treated differently.*

Example of valid variable names:
total
student_age
marks1

1.4.5 Scope and Lifetime of Variables

*In C programming, variables are used to store data that may be required during the execution of a program. However, the behavior of a variable in a program is determined not only by its value and data type but also by two important concepts known as **scope** and **lifetime**. These concepts define **where a variable can be accessed in a program** and **how long the variable remains in memory during the execution of the program**.*

Understanding scope and lifetime is essential for writing efficient and well-structured programs. These concepts help programmers control the accessibility of variables, prevent naming conflicts, and manage memory efficiently. By properly defining scope and lifetime, programmers can ensure that variables are used only where they are needed, which improves program clarity and maintainability.

Scope of a Variable

*The **scope of a variable** refers to the region or portion of a program in which the variable can be accessed or used. In other words, scope determines the **visibility of a variable** within the program. If a variable is declared in a particular section of the program, it may only be accessible within that section depending on its scope.*

*In C programming, the scope of a variable is mainly determined by the **location of its declaration**. Based on this, variables can have **local scope** or **global scope**.*

1. Local Scope (Local Variables)

*A **local variable** is a variable that is declared inside a function or within a block of code. The scope of a local variable is limited only to the function or block in which it is declared. This means that the variable cannot be accessed outside that particular function or block.*

Local variables are mainly used for performing calculations or storing temporary values that are needed only during the execution of a specific function. Because their scope is restricted, local variables help prevent interference between different parts of a program.

Example:

```
#include <stdio.h>

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

*In this example, the variable **x** is declared inside the main() function. Therefore, it can only be accessed within the main() function and cannot be used by other functions in the program.*

Local variables may also be defined inside blocks such as loops or conditional statements.

Example:

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

*In this case, the variable **i** exists only within the for loop and cannot be accessed outside it.*

2. Global Scope (Global Variables)

*A **global variable** is a variable that is declared outside all functions in a program. The scope of a global variable extends throughout the entire program, meaning that it can be accessed by any function within the program.*

Global variables are commonly used when multiple functions need to share or modify the same data. Since they are accessible from different parts of the program, they provide a convenient way for functions to communicate with each other.

Example:

```
#include <stdio.h>

int count = 10;   // Global variable

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

*In this example, the variable **count** is declared outside the main() function. Therefore, it can be accessed by any function in the program.*

Although global variables are useful, excessive use of them may make programs more difficult to understand and maintain. Therefore, programmers should use them carefully.

Lifetime of a Variable

*The **lifetime of a variable** refers to the duration for which the variable exists in the computer's memory during program execution. In other words, lifetime determines **how long a variable occupies memory before it is destroyed**.*

The lifetime of a variable depends on how and where the variable is declared in the program.

1. Lifetime of Local Variables

Local variables are created when the function or block in which they are declared begins execution. They remain in memory only while that function or block is being executed.

*Once the function completes its execution, the memory allocated for the local variable is automatically released. Therefore, local variables have a **short lifetime**, limited to the execution of the function.*

Example:

```
#include <stdio.h>

void display()
{
   int value = 20;   // Local variable
   printf("%d", value);
```

```
}

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

*In this example, the variable **value** is created when the display() function begins execution and is destroyed once the function finishes execution.*

2. Lifetime of Global Variables

Global variables have a longer lifetime compared to local variables. They are created when the program begins execution and remain in memory until the program terminates.

This means that global variables exist throughout the entire execution of the program and can be accessed by different functions whenever needed.

Example:

```
#include <stdio.h>

int total = 100;   // Global variable

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

*In this example, the variable **total** is created when the program starts running and remains in memory until the program ends.*

1.5 Data Types in C

*In the C programming language, **data types** specify the type of data that a variable can store and the type of operations that can be performed on that data. When a variable is declared in a program, the compiler must know the nature of the data it will store. This information is provided through the data type.*

Data types play a crucial role in programming because they help the compiler determine:

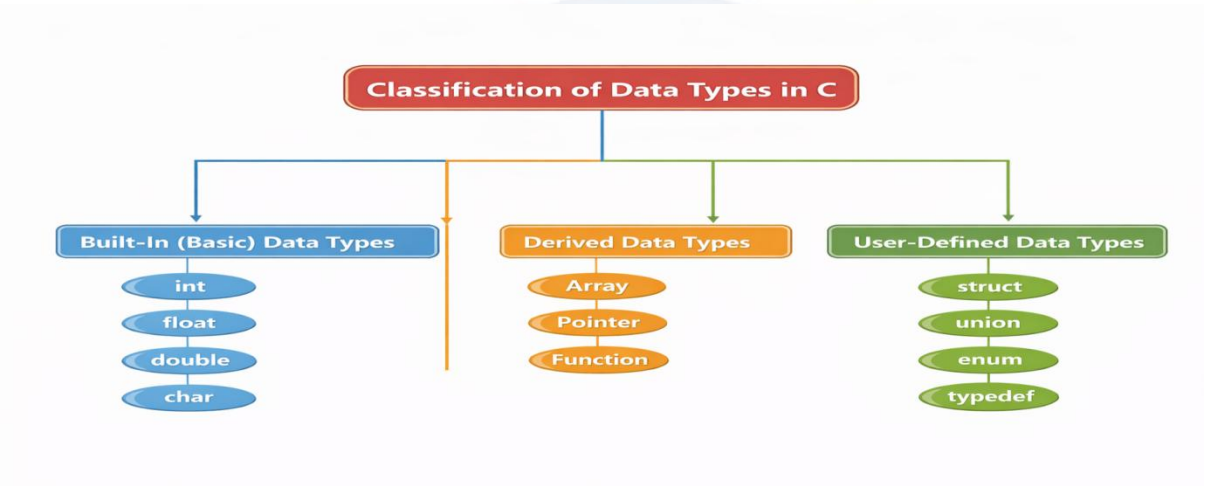
- *The **amount of memory** required to store the variable*
- *The **range of values** that can be stored*
- *The **operations that can be performed** on the data*

For example, integer values require different memory and operations compared to characters or decimal numbers. By specifying the appropriate data type, programmers ensure that the program uses memory efficiently and processes data correctly.

1.5.1 Types of Data Types in C

Data types in C are generally classified into three main categories:

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



1. Basic (Primary) Data Types

*Basic or primary data types are the **fundamental data types provided by the C language**. These data types are used to store simple and single values. They form the foundation upon which other complex data types are built.*

Each basic data type specifies the type of value that a variable can hold and determines the memory size allocated for that variable.

int Data Type

*The **int** data type is used to store integer values, which are whole numbers without any fractional or decimal part. It can represent positive numbers, negative numbers, and zero.*

*The size of an integer typically occupies **2 or 4 bytes of memory**, depending on the system architecture and compiler implementation.*

The int data type is widely used for counting operations, indexing elements in arrays, and controlling loops in programs.

Example:

int count = 10;

float Data Type

*The **float** data type is used to store floating-point numbers, which contain decimal or fractional parts. It is known as a **single-precision floating-point data type**.*

*A float variable generally occupies **4 bytes of memory** and provides approximately **6 decimal digits of precision**.*

This data type is commonly used in applications involving measurements, scientific calculations, and values that require decimal representation.

Example:

float price = 45.75;

double Data Type

*The **double** data type is used to store floating-point numbers with **double precision**. It provides greater accuracy than the float data type.*

*A double variable usually occupies **8 bytes of memory** and can store values with approximately **15 decimal digits of precision**.*

Because of its higher precision, double is often used in scientific calculations, financial applications, and programs that require more accurate decimal values.

Example:

double distance = 12345.6789;

char Data Type

*The **char** data type is used to store a **single character**, such as a letter, digit, or symbol.*

*A character variable occupies **1 byte of memory** and stores characters based on their **ASCII (American Standard Code for Information Interchange)** values.*

The char data type is commonly used in programs that involve character processing and string manipulation.

Example:

char grade = 'A';

Void Data Type

*The **void** data type represents the **absence of a value**. It is generally used when a function does not return any value.*

For example, if a function performs an action but does not produce a result, it is declared with the void data type.

Example:

void display();

2. Derived Data Types

Derived data types in C are data types that are created from the basic (primary) data types. They are formed by modifying or combining basic data types to represent more complex forms of data. Derived data types allow programmers to organize, store, and manipulate data in a more structured and efficient manner.

While basic data types such as int, float, char, and double store single values, derived data types help represent collections of data, memory addresses, or program modules. These data types provide flexibility in handling complex data structures and play an important role in developing large and modular programs.

Derived data types are called "derived" because they are constructed using existing basic data types. By using derived data types, programmers can create more powerful data structures that improve the efficiency and readability of programs.

*The most common **derived data types in C** include:*

- *Arrays*
- *Pointers*
- *Functions*

Each of these data types provides a different way of handling data and program logic.

Array

*An **array** is a derived data type that represents a collection of elements of the same data type stored in contiguous memory locations. All elements of an array share the same variable name but are distinguished by an index value.*

Arrays are useful when a program needs to store multiple values of the same type. Instead of creating many separate variables, a single array variable can be used to store all the values.

*The elements of an array are accessed using an index number, which usually starts from **0**.*

Example:

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

*In this example, the array **marks** stores five integer values representing marks.*

Arrays are widely used in programs that involve data processing, searching, sorting, and numerical computations.

Pointer

*A **pointer** is a special type of variable that stores the **memory address of another variable** rather than storing the actual value. Pointers provide direct access to memory locations and allow programs to manipulate data efficiently.*

Pointers are important in C programming because they enable dynamic memory allocation, efficient array handling, and communication between functions.

*To declare a pointer, the * symbol is used.*

Example:

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

In this example:

- *a is a variable that stores the value **10**.*
- *p is a pointer variable that stores the **memory address of a**.*

Pointers are commonly used in advanced programming concepts such as dynamic memory management and linked data structures.

Function

*A **function** is a block of code that performs a specific task. Functions are used to divide a large program into smaller and manageable parts. Each function performs a particular operation and may return a value to the calling program.*

Functions help improve program structure, readability, and reusability. By using functions, programmers can avoid repeating the same code multiple times.

Example:

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

*In this example, the function **add** takes two integer parameters and returns their sum.*

Functions play an important role in modular programming, where complex programs are divided into smaller modules for easier development and maintenance.

3. User-Defined Data Types

*In C programming, **user-defined data types** are data types that are created by programmers according to the requirements of a program. While the C language provides several built-in data types such as int, float, char, and double, these basic data types are sometimes not sufficient for representing complex data structures.*

User-defined data types allow programmers to create new types that can store multiple pieces of related information under a single name. This helps organize data in a structured way and improves the readability and maintainability of programs.

By using user-defined data types, programmers can represent real-world entities such as students, employees, products, or records more effectively. These data types make programs more flexible and help manage complex data more efficiently.

The main user-defined data types provided in C include:

- ***Structure (struct)***
- ***Union (union)***
- ***Enumeration (enum)***
- ***Typedef (typedef)***

Each of these data types serves a specific purpose in organizing and managing data.

Structure (struct)

*A **structure** is a user-defined data type that allows grouping of variables of different data types under a single name. Each member of a structure represents a different attribute of the data being represented.*

Structures are useful when we need to represent a collection of related information about an entity.

For example, a student record may contain different types of information such as student ID, name, and marks. These different pieces of information can be grouped together using a structure.

Syntax:

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

Example:

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

*In this example, the structure **student** contains three members: id, name, and marks.*

Union

*A **union** is similar to a structure in that it can store different data types. However, unlike a structure, all members of a union share the **same memory location**. This means that at any given time, only one member of the union can store a value.*

Unions are mainly used when memory efficiency is important.

Syntax:

```
union union_name
{
```

```
    data_type member1;
    data_type member2;
};
```

Example:

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

*In this example, the union **data** can store either an integer, a floating-point number, or a character, but only one value at a time.*

Enumeration (enum)

*An **enumeration** is a user-defined data type that consists of a set of named integer constants. It allows programmers to assign meaningful names to constant values, which improves the readability of programs.*

Enumerations are useful when a variable can take one value from a predefined set of values.

Syntax:

```
enum enum_name {value1, value2, value3};
```

Example:

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

In this example, the values MON, TUE, WED, THU, and FRI represent integer constants.

Typedef

*The **typedef** keyword is used to create an alternative name (alias) for an existing data type. It does not create a new data type but simply provides another name for an existing type.*

Typedef is useful when dealing with complex data type declarations because it makes the code easier to read and understand.

Syntax:

```
typedef existing_data_type new_name;
```

Example:

typedef int number;
number a = 10;

In this example, ***number*** *becomes an alternative name for the data type int.*

1.5.2 Type Conversion in C

In the C programming language, ***type conversion*** *is the process of converting a value from one data type into another data type. Since programs often involve multiple kinds of data such as integers, characters, and floating-point numbers, it becomes necessary to convert data from one type to another in order to perform operations correctly.*

Every variable in C has a specific data type that determines the ***kind of value it can store****, the* ***amount of memory allocated****, and the* ***operations that can be performed on it****. When different data types appear together in an expression, the C compiler must ensure that the data types are compatible before performing any operation. This compatibility is achieved through type conversion.*

Type conversion ensures that arithmetic expressions are evaluated properly and that the result produced by the program is accurate. Without type conversion, operations involving different data types may lead to incorrect results, loss of precision, or compilation errors.

For example, when an integer value is added to a floating-point number, the integer must be converted into a floating-point value so that the operation can be performed correctly.

Example:

int a = 10;
float b = 2.5;

float result = a + b;

In this case, the integer value ***a*** *is converted into a floating-point value before the addition operation is performed.*

Need for Type Conversion

Type conversion is required in programming for several important reasons:

- *To allow operations between* ***different types of data***
- *To maintain* ***accuracy and precision*** *in calculations*

- *To prevent **data type mismatch errors***
- *To ensure correct **evaluation of expressions***
- *To make programs more **flexible and efficient***

When two operands of different types are used in an expression, the compiler automatically adjusts the data types to perform the operation correctly.

Types of Type Conversion in C

Type conversion in C can be classified into two main categories:

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

1. Implicit Type Conversion (Automatic Conversion)

Implicit type conversion refers to the automatic conversion of one data type into another performed by the compiler. This conversion takes place when an expression contains operands of different data types.

*During implicit conversion, the compiler converts the **lower-precision data type into a higher-precision data type** so that the operation can be performed without loss of data.*

*This process is also called **type promotion**.*

Example:

```
int a = 5;
float b = 4.5;

float result = a + b;
```

*Here, the integer value **a** is automatically converted into a floating-point value before the addition is performed.*

Example:

```
char c = 'A';
int num = 10;

int result = c + num;
```

In this example, the character c is converted into its ASCII integer value before the addition operation.

2. Explicit Type Conversion (Type Casting)

*Explicit type conversion, also known as **type casting**, occurs when the programmer manually converts a value from one data type to another.*

In this method, the programmer explicitly instructs the compiler to perform the conversion using a specific syntax. This allows greater control over how data is converted.

*Type casting is particularly useful when the programmer wants to **change the type of a variable temporarily during an operation**.*

Syntax of Type Casting

(data_type) expression

Here, the expression is converted into the specified data type.

Example of Type Casting

```
int a = 10;
int b = 3;

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

*In this example, the integer variable **a** is converted into a floating-point value before performing division. Without this conversion, the division of two integers would produce an integer result.*

Example Program

```
#include <stdio.h>

int main()
{
   int a = 7;
   int b = 2;

   float result = (float)a / b;

   printf("Result = %f", result);
```

```
    return 0;
}
```

Output:

Result = 3.500000

Here, the type casting ensures that the result contains a decimal value.

1.6 Input and Output Functions in C

Input and Output (I/O) functions in C are used to enable communication between a program and the outside world. A program often needs to ***receive data from the user*** *and* ***display processed results back to the user****. These operations are performed using input and output functions.*

Input refers to the process of ***accepting data from input devices*** *such as the keyboard, while output refers to the process of* ***displaying information on output devices*** *such as the monitor.*

In C programming, standard input and output operations are performed using functions provided in the ***standard input-output library****, which is included through the header file:*

#include <stdio.h>

This library provides various functions that allow programs to read data from the keyboard and display information on the screen.

Internally, computers process data in ***binary form****, but users interact with programs using* ***human-readable characters and numbers****. Input and output functions therefore perform the task of* ***converting data between the internal binary representation used by the computer and the readable form used by humans****.*

Classification of Input and Output Functions in C

Input and output functions in C are broadly classified into two categories based on whether they use formatting control or not.

1. ***Non-Formatted Input/Output Functions***
2. ***Formatted Input/Output Functions***

1.6.1. Non-Formatted Input/Output Functions

Non-formatted I/O functions are simple input/output functions that ***do not use format specifiers****. These functions read or display data in its basic form without allowing the programmer to control the formatting of the data.*

*These functions are generally used for **simple character or string operations**.*

Characteristics of non-formatted I/O functions:

- *They do not require format specifiers such as %d, %f, or %c.*
- *They are simple and easy to use.*
- *They handle basic character or string input/output.*
- *They are suitable for simple programs or character-based processing.*

Common examples include:

- *getchar()*
- *putchar()*
- *gets()*
- *puts()*

Classification of Non-Formatted I/O Functions

Non-formatted I/O functions can be further classified into two categories:

A. Character Input/Output Functions

*These functions read or write **a single character at a time**.*

Examples:

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

B. String Input/Output Functions

*These functions handle **strings (a sequence of characters)**.*

Examples:

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

A. Character Input/Output Functions

getchar() Function

*The getchar() function is a **non-formatted input function** used to read a single character from the standard input device, usually the keyboard.*

*This function reads exactly **one character at a time**, including whitespace characters such as spaces, tabs, and newline characters. The function waits until the user presses the **Enter key**, after which the first entered character is returned.*

The value returned by getchar() is typically stored in a variable of type char.

*The getchar() function is commonly used in programs that require **character-by-character input**, such as menu-driven programs, password validation, or text processing applications.*

Syntax

```
char variable;
variable = getchar();
```

Example

```
#include <stdio.h>

int main()
{
   char ch;

   printf("Enter a character: ");
   ch = getchar();

   printf("You entered: %c", ch);

   return 0;
}
```

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

putchar() Function

*The putchar() function is a **non-formatted output function** used to display a single character on the standard output device (usually the screen).*

It prints one character at a time. The character to be printed can be:

- *A character constant*
- *A variable of type char*
- *The result of a character expression*

The putchar() function is simple and efficient and is often used in programs that display characters inside loops or when printing characters individually.

Syntax

```
putchar(character);
```

Example

```
#include <stdio.h>

int main()
{
   char ch = 'A';

   putchar(ch);

   return 0;
}
```

In this example, the function putchar() displays the character stored in the variable ch on the screen.

B. String Input / Output Functions

*String input/output functions are used to **read and display a sequence of characters**, known as a string. A string is typically stored in a **character array** and is terminated by a special character called the **null character ('\0')**.*

The main string I/O functions are:

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

gets() Function

*The gets() function is a **non-formatted input function** used to read a string from the keyboard. It reads characters continuously until the user presses the **Enter key**.*

*The entire line of input is stored in a character array, and the function automatically adds a **null character ('\0')** at the end of the string to indicate the end of the string.*

*One advantage of gets() is that it allows input containing **spaces**, which makes it useful for reading full sentences or names.*

*However, the gets() function **does not perform boundary checking**. If the user enters more characters than the array can hold, it may cause **buffer overflow**, which can lead to program errors or security issues.*

Because of this limitation, gets() has been removed from modern C standards, but it is still sometimes studied 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, the function gets() reads a complete line of text and stores it in the array name.

puts() Function

*The puts() function is a **non-formatted output function** used to display a string on the 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, the puts() function automatically moves the cursor to the next line by inserting a newline character.

Because of this feature, puts() is convenient for displaying text messages without manually adding the newline character (\n).

Example

```
#include <stdio.h>

int main()
{
   char msg[] = "Welcome to C Programming";

   puts(msg);

   return 0;
}
```

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

1.6.2. Formatted Input / Output Functions

*Formatted input/output functions in C are used to **read and display data in a specified format**. These functions allow programmers to control how data is interpreted during input and how it appears when displayed on the screen. By using **format specifiers**, formatted I/O functions enable the program to handle various types of data such as integers, floating-point numbers, characters, and strings efficiently.*

*Unlike non-formatted I/O functions, formatted functions allow the programmer to define **the exact format in which data should be read or printed**. This makes them highly useful in programs that require structured input or well-organized output.*

*All formatted input and output functions in C are provided through the **standard input/output library**, which is included using the header file:*

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

Formatted I/O functions are widely used in almost every C program because they allow accurate reading of data and neat presentation of output.

Types of Formatted I/O Functions

Formatted input/output functions in C are mainly divided into two categories:

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

scanf() Function

*The scanf() function is a **formatted input function** used to read data from the standard input device, usually the keyboard. It allows the programmer to accept different types of input values by using appropriate **format specifiers**.*

*The function reads the input according to the specified format and stores the values in the memory locations of the variables provided as arguments. Since the function needs the memory addresses of variables, the **address-of operator (&)** is used before variable names (except for strings).*

If the input provided by the user does not match the specified format specifier, the function stops reading further input, which may result in incorrect program behavior.

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 program, scanf() reads an integer value for **age** and a floating-point value for **marks**. The values are stored in the corresponding variables using their memory addresses.*

Common Format Specifiers Used in scanf()

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

Format specifiers instruct the compiler how to interpret the data being entered.

printf() Function

*The printf() function is a **formatted output function** used to display data on the screen. It allows the programmer to print different types of values in a controlled and organized format.*

*The function converts the internal binary representation of data stored in memory into a **human-readable format** and displays it on the output device. Format specifiers within the format string determine how each value is printed.*

*The printf() function also supports **escape sequences**, **field width**, and **precision control**, which help produce clear and well-formatted 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:

- *%d is used to print an integer value.*
- *%.2f prints a floating-point value with two digits after the decimal point.*

Common Format Specifiers Used in printf()

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

These format specifiers allow the programmer to control how the data appears on the screen.

Escape Sequences in C

Escape sequences are special character combinations used within strings to represent ***non-printable characters or formatting instructions****. They begin with a* ***backslash (\)*** *followed by a specific character.*

Escape sequences are commonly used with output functions like printf() to control how text is displayed.

Purpose of Escape Sequences

Escape sequences help programmers to:

- *Move the cursor to a new line or tab position*
- *Insert special characters into output*
- *Control the layout and formatting of displayed text*
- *Improve readability of program output*

Common Escape Sequences in C

Escape Sequence	***Meaning***	***Usage***
\n	*New line*	*Moves cursor to next line*
\t	*Horizontal tab*	*Inserts tab space*

\b	Backspace	Moves cursor one position back
\r	Carriage return	Moves cursor to beginning of line
\f	Form feed	Page break
\\	Backslash	Prints backslash
\'	Single quote	Prints single quotation mark
\"	Double quote	Prints double quotation mark
\a	Alert	Produces beep sound
\0	Null character	Indicates end of string

4. Operators in C

Operators in C are special symbols used to perform various operations on variables and values. They instruct the compiler to perform specific tasks such as arithmetic calculations, comparisons, logical evaluations, and assignments.

*In programming, the data stored in variables often needs to be processed, compared, or modified. Operators provide the mechanism to manipulate this data and control the behavior of the program. Every operator works with one or more **operands**, which are the values or variables on which the operation is performed.*

For example:

int sum = a + b;

Here:

- *+ is the **operator***
- *a and b are **operands***

Operators are fundamental to programming because they allow programs to perform calculations, evaluate conditions, and update variable values during execution.

Major Types of Operators in C

C provides several categories of operators. The most commonly used operators include:

1. *Arithmetic Operators*
2. *Relational Operators*
3. *Logical Operators*
4. *Assignment Operators*
5. *Increment and Decrement Operators*

Each of these operators performs a specific type of operation.

1. Arithmetic Operators

Arithmetic operators are used to perform mathematical calculations on numeric values. These operators are commonly used in programs that involve numerical computations such as addition, subtraction, multiplication, and division.

Arithmetic operators work with numeric data types such as ***int, float, and double****.*

Arithmetic Operators Table

Operator	***Operation***	***Description***	***Example***
+	*Addition*	*Adds two operands*	*a + b*
-	*Subtraction*	*Subtracts second operand from first*	*a - b*
*	*Multiplication*	*Multiplies two operands*	*a * b*
/	*Division*	*Divides one operand by another*	*a / b*
%	*Modulus*	*Returns remainder after division*	*a % b*

Example

```
#include <stdio.h>

int main()
{
   int a = 10;
   int b = 3;

   printf("%d\n", a + b); // 13
   printf("%d\n", a % b); // 1

   return 0;
}
```

Arithmetic operators are essential for performing numerical calculations in programs.

2. Relational Operators

Relational operators are used to compare two values. The result of a relational expression is either ***true (1)*** *or* ***false (0)****.*

These operators are commonly used in ***conditional statements and loops****.*

Relational Operators Table

Operator	***Description***	***Example***	***Result***
==	*Equal to*	*a == b*	*true if values are equal*
!=	*Not equal to*	*a != b*	*true if values are different*

>	Greater than	a > b	true if a is greater
<	Less than	a < b	true if a is smaller
>=	Greater than or equal to	a >= b	true if condition satisfied
<=	Less than or equal to	a <= b	true if condition satisfied

Example

```
#include <stdio.h>

int main()
{
   int a = 10;
   int b = 5;

   printf("%d\n", a > b);  // 1 (true)
   printf("%d\n", a == b); // 0 (false)

   return 0;
}
```

Relational operators help programs make decisions based on comparisons.

3. Logical Operators

Logical operators are used to combine multiple conditions or expressions. They are mainly used in decision-making statements such as ***if statements, loops, and conditional expressions****.*

Logical operators operate on Boolean values.

Logical Operators Table

Operator	***Name***	***Description***	***Example***
&&	Logical AND	Returns true if both conditions are true	(a>b)&&(a>c)
||	Logical OR	Returns true if at least one condition is true	(a>b)||(b>c)
!	Logical NOT	Reverses the logical value	!(a>b)

Example

```
#include <stdio.h>

int main()
{
   int a = 10;
   int b = 5;
```

```
    printf("%d\n", (a > b) && (b > 2));
    printf("%d\n", (a < b) || (b > 2));

    return 0;
}
```

Logical operators allow programs to evaluate multiple conditions simultaneously.

4. Assignment Operators

Assignment operators are used to assign values to variables. The basic assignment operator assigns the value on the right-hand side to the variable on the left-hand side.

C also provides ***compound assignment operators****, which combine arithmetic operations with assignment.*

Assignment Operators Table

Operator	*Description*	*Example*
=	*Assign value*	*a = 10*
+=	*Add and assign*	*a += 5*
-=	*Subtract and assign*	*a -= 5*
*=	*Multiply and assign*	*a *= 5*
/=	*Divide and assign*	*a /= 5*
%=	*Modulus and assign*	*a %= 5*

Example

```
#include <stdio.h>

int main()
{
    int a = 10;

    a += 5;
    printf("%d", a); // 15

    return 0;
}
```

Assignment operators simplify programming by reducing repeated expressions.

5. 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.

Increment and Decrement Operators Table

Operator	***Name***	***Description***	***Example***
++	*Increment*	*Increases value by 1*	*a++*
--	*Decrement*	*Decreases value by 1*	*a--*

Types of Increment

Pre-Increment

The value is increased before it is used.

++a;

Post-Increment

The value is increased after it is used.

a++;

Example

```
#include <stdio.h>

int main()
{
   int a = 5;

   a++;
   printf("%d\n", a); // 6

   a--;
   printf("%d\n", a); // 5

   return 0;
}
```

These operators are very useful for iteration and counting operations in loops.

6. Separators (Punctuators) in C

Separators are special characters used to organize the structure of a program. They separate statements, define blocks of code, and group expressions.

Common separators in C include:

; () {} [] , .

Examples

Separator	***Usage***
;	*Ends a statement*
{ }	*Defines a block of code*
( )	*Used in function calls*
[ ]	*Used in arrays*
,	*Separates variables*

Separators help maintain proper program syntax and structure.

Bitwise Operators in C

*Bitwise operators in C are special operators that perform operations directly on the **binary representation of integers**. Unlike arithmetic or logical operators that work on whole numbers, bitwise operators operate on **individual bits (0 and 1)** of the data stored in memory.*

*Since computers internally represent numbers in **binary form**, bitwise operators allow programmers to manipulate these binary digits directly. These operators are mainly used in **system programming, embedded systems, device drivers, data compression, and performance optimization**, where efficient manipulation of bits is required.*

*Bitwise operators are generally used with **integer data types**, such as int, char, and long.*

*For example, the decimal number **5** is stored in binary form as:*

5 = 00000101

Bitwise operators manipulate these bits to produce a new result.

Types of Bitwise Operators in C

C provides the following bitwise operators:

Operator	***Name***
&	*Bitwise AND*
|	*Bitwise OR*
^	*Bitwise XOR*

~	*Bitwise NOT*
<<	*Left Shift*
>>	*Right Shift*

1. Bitwise AND Operator (&)

*The **bitwise AND operator** compares each bit of two numbers and performs the AND operation.*

*The result is **1 only when both bits are 1**. Otherwise, the result is 0.*

Example

a = 5 → 00000101
b = 3 → 00000011
----
a & b → 00000001 → 1

Program

```
#include <stdio.h>

int main()
{
   int a = 5, b = 3;
   printf("%d", a & b);
   return 0;
}
```

Output:

1

*The AND operator is commonly used to **mask specific bits** in a number.*

2. Bitwise OR Operator (|)

*The **bitwise OR operator** compares corresponding bits of two operands.*

*The result becomes **1 if at least one of the bits is 1**.*

Example

a = 5 → 00000101
b = 3 → 00000011

---

a | b → 00000111 → 7

Program

```
#include <stdio.h>

int main()
{
   int a = 5, b = 3;
   printf("%d", a | b);
   return 0;
}
```

Output:

7

The OR operator is often used to **set specific bits in a number**.

3. Bitwise XOR Operator (^)

The **bitwise XOR (exclusive OR)** operator compares each pair of bits.

The result is **1 if the bits are different** and **0 if they are the same**.

Example

a = 5 → 00000101
b = 3 → 00000011

---

a ^ b → 00000110 → 6

Program

```
#include <stdio.h>

int main()
{
   int a = 5, b = 3;
   printf("%d", a ^ b);
   return 0;
}
```

Output:

6

XOR is commonly used in ***encryption algorithms and bit toggling operations****.*

4. Bitwise NOT Operator (~)

The ***bitwise NOT operator*** *is a unary operator that* ***inverts every bit*** *of the operand.*

All ***0s become 1s****, and all* ***1s become 0s****.*

Example

a = 5 ⟶ 00000101
~a ⟶ 11111010

Because computers use ***two's complement representation****, the result may appear as a negative number.*

Program

```
#include <stdio.h>

int main()
{
   int a = 5;
   printf("%d", ~a);
   return 0;
}
```

5. Left Shift Operator (<<)

The ***left shift operator*** *shifts the bits of a number toward the left.*

Zeros are inserted into the rightmost positions.

Each left shift operation usually ***multiplies the number by 2****.*

Example

a = 5 ⟶ 00000101
a << 1 ⟶ 00001010 ⟶ 10

Program

```
#include <stdio.h>

int main()
{
   int a = 5;
   printf("%d", a << 1);
   return 0;
}
```

Output:

10

6. Right Shift Operator (>>)

*The **right shift operator** shifts the bits of a number toward the right.*

*Each right shift operation usually **divides the number by 2**.*

Example

a = 8 → 00001000
a >> 1 → 00000100 → 4

Program

```
#include <stdio.h>

int main()
{
   int a = 8;
   printf("%d", a >> 1);
   return 0;
}
```

Output:

4

Bitwise Operators in C

*Bitwise operators operate directly on the **binary representation of integers**. Each bit of one operand is compared with the corresponding bit of another operand, and the operation is performed bit by bit.*

Bitwise Operators Truth Table

A	*B*	*A & B (AND)*	*A | B (OR)*	*A ^ B (XOR)*
0	*0*	*0*	*0*	*0*
0	*1*	*0*	*1*	*1*
1	*0*	*0*	*1*	*1*
1	*1*	*1*	*1*	*0*

Operator	***Name***	***Operation Logic***	***Binary Example***	***Result***
&	*Bitwise AND*	*Result bit is **1 only if both bits are 1***	*0101 & 0011 = 0001*	*1*
|	*Bitwise OR*	*Result bit is **1 if at least one bit is 1***	*0101 | 0011 = 0111*	*7*
^	*Bitwise XOR*	*Result bit is **1 if bits are different***	*0101 ^ 0011 = 0110*	*6*
~	*Bitwise NOT*	***Inverts all bits** (0 becomes 1, 1 becomes 0)*	*~0101 = 1010*	*-6**
<<	*Left Shift*	*Bits move **left**, zeros fill right side*	*0101 << 1 = 1010*	*10*
>>	*Right Shift*	*Bits move **right**, leftmost bits filled depending on sign*	*1000 >> 1 = 0100*	*4*

Conditional Operator

*The **conditional operator** in C is a shorthand operator used to make simple decisions in a program. It is also known as the **ternary operator** because it operates on **three operands**. This operator is often used as a compact alternative to the if-else statement.*

The conditional operator evaluates a condition and returns one of two values depending on whether the condition is true or false. It helps simplify code by reducing the need for multiple lines of conditional statements.

The general structure of the conditional operator consists of three parts:

1. *A **condition** that is evaluated first*
2. *An **expression executed if the condition is true***
3. *An **expression executed if the condition is false***

Syntax

condition ? expression1 : expression2;

Where:

- ***condition** → A logical expression that is evaluated first*
- ***expression1** → Executed if the condition is true*
- ***expression2** → Executed if the condition is false*

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 first.*
- *If the condition is **true**, the value of a is assigned to max.*
- *If the condition is **false**, the value of b is assigned to max.*

In this case, since b is greater than a, the program prints:

Special Operators in C

Special operators in C are operators that perform specific and specialized tasks related to ***memory handling, address manipulation, structure access, and expression evaluation****. Unlike arithmetic or logical operators that perform calculations or comparisons, special operators provide additional capabilities that help programmers manage memory and access complex data structures efficiently.*

These operators are particularly important in system-level programming because they allow programmers to interact directly with memory locations and control how data is stored and accessed. Special operators are widely used when working with ***pointers, arrays, and structures****, which are essential concepts in the C programming language.*

The main special operators provided in C include:

- ***sizeof operator***
- ***Comma operator (,)***
- **Pointer operators (& and)*
- ***Member access operators (. and ->)***

Each of these operators performs a unique function and contributes to the flexibility and power of the C language.

1. sizeof Operator

The ***sizeof operator*** *is used to determine the amount of memory required to store a variable or a particular data type. It returns the size in bytes that a variable occupies in the computer's memory.*

This operator is useful when working with arrays, structures, and dynamic memory allocation, because it helps programmers understand how much memory is needed for storing data.

For example, different data types require different amounts of memory. The sizeof operator allows programmers to check the memory size of data types such as int, float, or char.

Thus, the sizeof operator plays an important role in efficient memory management in C programs.

2. Comma Operator (,)

*The **comma operator** is used to combine multiple expressions in a single statement. When several expressions are separated by commas, they are evaluated from **left to right**, and the value of the final expression becomes the result of the entire expression.*

This operator is useful when a programmer needs to perform multiple operations within a single statement. It is commonly used in loops and complex expressions where multiple operations must be evaluated sequentially.

Although it is not frequently used in simple programs, it can simplify certain expressions and make code more compact.

3. Pointer Operators (& and)

*Pointer operators are special operators used to work with **memory addresses**.*

*The **address-of operator (&)** is used to obtain the memory address of a variable. Every variable stored in memory has a unique address, and this operator allows the programmer to access that address.*

*The **dereference operator (*)** is used to access the value stored at a specific memory address. When a pointer variable stores the address of another variable, the dereference operator allows the programmer to retrieve or modify the value stored at that location.*

*Pointer operators are essential in C programming because they allow efficient memory usage and are widely used in advanced programming concepts such as **dynamic memory allocation, arrays, and structures**.*

4. Member Access Operators (. and ->)

*Member access operators are used to access the members of **structures and unions**.*

*The **dot operator (.)** is used to access members of a structure when the structure variable itself is used directly.*

*The **arrow operator (->)** is used to access members of a structure when a pointer to the structure is used.*

These operators allow programmers to work with complex data structures that contain multiple variables grouped together.

Structures are commonly used to represent real-world entities such as student records, employee information, or product details, and member access operators make it possible to access individual elements of these structures.

Special Operators in C

Operator	Name	Purpose / Description	Example Code	Output
sizeof	Sizeof Operator	Determines the amount of memory (in bytes) occupied by a variable or data type.	printf("%lu", sizeof(int));	4
,	Comma Operator	Allows multiple expressions to be evaluated in one statement. The last expression gives the final value.	int a,b; a=(b=5,b+10); printf("%d",a);	15
&	Address-of Operator	Returns the memory address of a variable.	int a=10; printf("%p",&a);	Memory address of a
*	Pointer (Dereference) Operator	Retrieves the value stored at the memory address pointed by a pointer.	int a=10; int *p=&a; printf("%d",*p);	10
.	Dot Operator	Accesses members of a structure using a structure variable.	struct s{int id;}; struct s st; st.id=101; printf("%d",st.id);	101
->	Arrow Operator	Accesses members of a structure using a pointer to a structure.	struct s{int id;}; struct s st; struct s *p=&st; p->id=202; printf("%d",p->id);	202

2.Control Statements in C

2.1.1 Introduction

Control statements in C are used to ***control the flow of execution*** *of a program. Normally, a C program executes statements sequentially, meaning that statements are executed one after another in the order in which they appear. However, many real-world problems require a program to* ***make decisions, repeat certain tasks, or skip certain parts of code depending on conditions****.*

Control statements provide this flexibility by allowing the program to change its normal execution order. They help implement ***decision-making, repetition, and branching logic*** *in programs.*

For example, a program may need to:

- *Check whether a student has passed or failed*

- *Repeat a calculation several times*
- *Choose one operation from several options*
- *Skip certain statements under specific conditions*

*Control statements make such operations possible and therefore form a **fundamental part of program logic and structure**.*

Classification of Control Statements

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

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

Each category controls program execution in a different way.

1. Selection (Decision-Making) Statements

*Selection statements are used to **choose one block of code for execution from multiple alternatives** based on a specified condition.*

These statements evaluate a condition and determine which set of instructions should be executed.

The main selection statements in C include:

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

These statements enable programs to make logical decisions similar to human decision-making.

2.1.2 if Statement

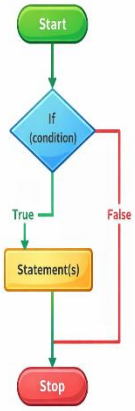
*The **if statement** is the simplest decision-making control statement in C. It allows a block of code to execute **only when a specified condition is true**.*

*If the condition evaluates to **true (non-zero)**, the statements inside the if block are executed. If the condition evaluates to **false (zero)**, the block is skipped and the program continues with the next statement.*

*The if statement is mainly used for **simple decision-making situations**.*

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 program:

- *The condition marks >= 40 is evaluated.*
- *Since the condition is **true**, the statement inside the if block is executed.*
- *Therefore, the message **"Student is Passed"** is displayed.*

If the marks were less than 40, the statement inside the if block would not execute.

Key Points

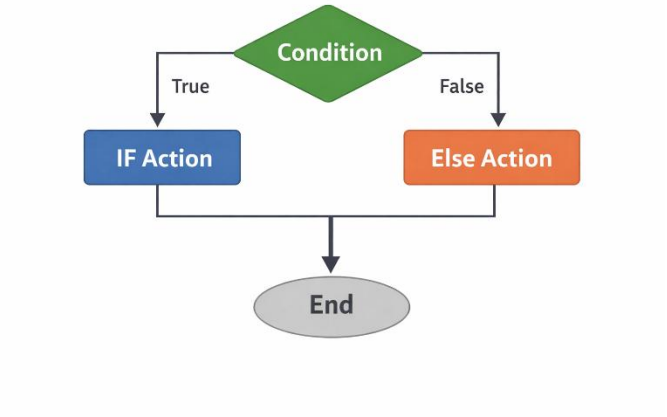
- *Executes statements only when the condition is true*
- *Condition must be written inside parentheses ( )*
- *Used for simple decision-making*
- *Works with relational and logical operators*

2.1.3 if–else Statement

*The **if–else statement** is used when a program needs to choose between **two alternative actions**.*

*If the condition is **true**, the statements in the if block are executed. If the condition is **false**, the statements in the else block are executed.*

*This ensures that **exactly one block of code is executed**.*



Syntax

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

In this example:

- *The condition marks >= 40 is evaluated.*
- *Since the condition is **false**, the else block executes.*
- *Therefore, the output is **"Student is Failed"**.*

Key Points

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

2.1.4 Nested if Statement

*A **nested if statement** occurs when an if statement is placed **inside another if or else block**.*

*This structure is used when multiple conditions must be checked in a **hierarchical or multi-level manner**. The inner condition is evaluated **only if the outer condition is true**.*

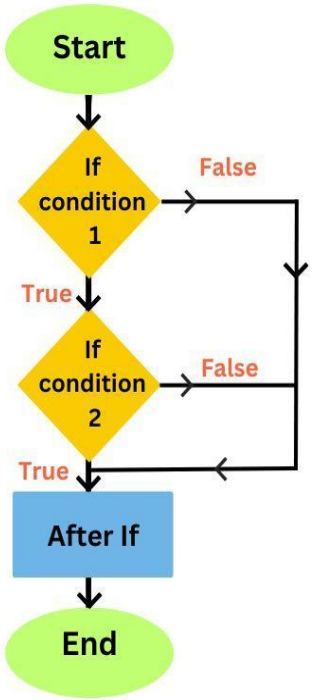
*Nested if statements are useful in programs that require **multi-level decision-making**, such as grading systems, eligibility checks, and classification problems.*

Syntax

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

Syntax 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

1. *First the program checks whether marks >= 40.*
2. *If this condition is true, the program evaluates the second condition marks >= 75.*
3. *If both conditions are true, the output is* ***"Distinction"****.*
4. *If the first condition is true but the second condition is false, the output is* ***"Pass"****.*
5. *If the first condition is false, the output is* ***"Fail"****.*

Key Points

- *Used for multi-level decision-making*
- *Inner if executes only when outer if is true*
- *Improves logical structuring of complex conditions*
- *Proper indentation improves readability*

2.1.5 switch Statement

*The **switch statement** is another decision-making control statement in C. It is used when a program must choose one block of code from **multiple possible options**.*

*Instead of using long if–else if ladders, the switch statement allows the program to evaluate an expression and match its value with predefined **case labels**.*

If a match is found, the corresponding block of code 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 contain **constant values***
- *break prevents execution from continuing to the next case*
- *default executes when no case matches*

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 of other cases.*
- *If no case matches, the default block is executed.*

Advantages of switch Statement

- *Improves readability of programs*
- *Easier to maintain compared to long if–else ladders*
- *Useful when comparing a variable with many constant values*

Limitations of switch Statement

- *Works only with constant values*
- *Cannot evaluate relational expressions*
- *Cannot directly check ranges of values*

2.2. Iterative (Looping) Statements

*Iterative statements, also known as **looping statements**, are used to execute a block of code repeatedly as long as a specified condition remains true. In many programs, certain tasks must be performed multiple times. Writing the same statements repeatedly would make the program lengthy and inefficient. Looping statements solve this problem by allowing a set of instructions to be executed automatically until a particular condition becomes false.*

In C programming, looping statements help reduce code redundancy and improve program efficiency. They are widely used in situations such as ***processing lists of data, performing repeated calculations, generating sequences of numbers, and handling menu-driven programs****.*

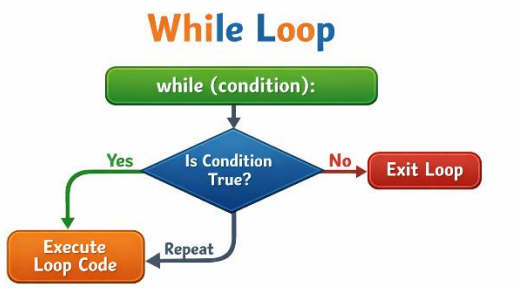
C provides three main looping statements:

- ***while loop***
- ***do–while loop***
- ***for loop***

2.2.1. while Loop

The ***while loop*** *is an iterative control statement used to repeatedly execute a block of statements as long as a specified condition evaluates to true. It is called an* ***entry-controlled loop*** *because the condition is checked before the execution of the loop body.*

If the condition is false initially, the loop body will not execute even once.



Syntax

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

Working of while Loop

1. *The condition is evaluated first.*
2. *If the condition is* ***true****, the statements inside the loop are executed.*
3. *After executing the loop body, control returns to the condition again.*
4. *The loop continues until the condition becomes* ***false****.*
5. *When the condition becomes false, the loop terminates and control moves to the next statement.*

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 of while Loop

- *It is an **entry-controlled loop**.*
- *The condition is checked before executing the loop body.*
- *The loop body may not execute if the condition is false initially.*
- *The loop variable must be updated inside the loop.*

Advantages

- *Simple and easy to use.*
- *Suitable when the number of iterations is not known in advance.*
- *Provides flexible loop control based on conditions.*

Limitations

- *May create an **infinite loop** if the condition is not updated properly.*
- *Not ideal when the number of iterations is fixed.*

2.2.2 do–while Loop

*The **do–while loop** is another iterative control statement used to execute a block of statements repeatedly until a given condition becomes false. It is known as an **exit-controlled loop** because the condition is checked after executing the loop body.*

*This means that the loop body **executes at least once**, even if the condition is false at the beginning.*

Syntax

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

Note: *The semicolon (;) after the condition is mandatory.*

Working of do–while Loop

1. *The statements inside the loop body are executed first.*
2. *After executing the loop body, the condition is evaluated.*
3. *If the condition is **true**, the loop repeats.*
4. *If the condition is **false**, the loop terminates.*

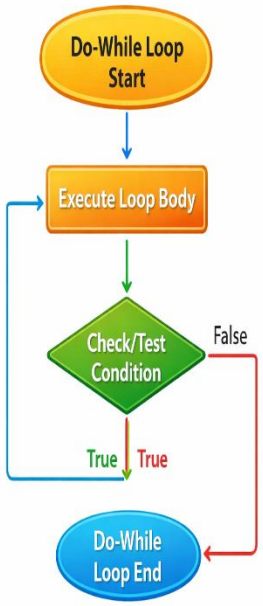
Example

```
#include <stdio.h>

int main()
{
   int i = 1;

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

   return 0;
}
```



Output

1 2 3 4 5

Key Points of do–while Loop

- *It is an **exit-controlled loop**.*
- *The loop body executes **at least once**.*
- *The condition is checked after executing the loop body.*
- *A semicolon after while(condition) is required.*

Advantages

- *Guarantees at least one execution of the loop body.*
- *Useful in **menu-driven programs**.*
- *Suitable when the first execution must occur before checking the condition.*

Limitations

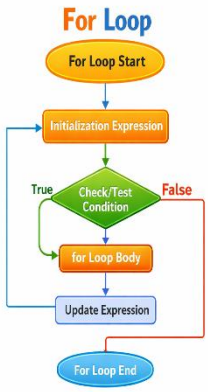
- *May lead to unnecessary execution if the condition is false initially.*
- *Risk of infinite loops if the condition is not updated.*

2.2.3 for Loop

*The **for loop** is an iterative control statement used to execute a block of statements repeatedly for a specific number of times. It is an **entry-controlled loop** in which the condition is evaluated before the loop body executes.*

*The for loop is particularly useful 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***
   Determines whether the loop should continue executing.
3. ***Increment / Decrement***
   Updates the loop variable after each iteration.

Working of for Loop

1. *Initialization is executed once at the beginning.*
2. *The condition is evaluated.*
3. *If the condition is **true**, the loop body executes.*
4. *After execution, the increment or decrement operation is performed.*
5. *Control returns to the condition.*
6. *The loop continues until the 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 of for Loop

- *It is an **entry-controlled loop**.*
- *Initialization, condition, and increment are written in a single statement.*
- *All three components are optional.*
- *It is best suited when the number of iterations is known.*

Advantages

- *Compact and structured syntax.*
- *Easy to understand and maintain.*
- *Ideal for counting and sequence generation.*

Limitations

- *Less flexible for complex conditions.*
- *Incorrect conditions may lead to infinite loops.*

Concept	***while Loop***	***do–while Loop***	***for Loop***
Basic Idea	*Executes a block of code repeatedly **while a condition remains true***	*Executes the block of code **first**, then checks the condition*	*Executes a block of code **a specific number of times** using a loop counter*
Control Mechanism	*Entry-controlled loop*	*Exit-controlled loop*	*Entry-controlled loop*
Condition Evaluation	*Condition is checked **before execution***	*Condition is checked **after execution***	*Condition is checked **before execution***
Execution Guarantee	*Loop may **not execute at all** if the condition is false initially*	*Loop **executes at least once** even if condition is false*	*Loop may **not execute** if condition is false initially*
Iteration Knowledge	*Used when the **number of repetitions is not known in advance***	*Used when the **task must be performed at least once***	*Used when the **number of repetitions is known***
Structure Concept	*Initialization and update are usually written **separately from the loop statement***	*Similar to while loop but **condition appears at the end***	*Initialization, condition, and update are written **in one statement***
Typical Usage	*Condition-based repetition*	*Menu-driven programs and*	*Counting loops and fixed iterations*

		validation loops	

2.3 Jump Statements in C

Jump statements are control statements that allow the program to ***transfer control immediately from one point of the program to another point*** *without following the normal sequential execution of statements. In a typical program, statements are executed in the order in which they appear. However, in many situations, it becomes necessary to* ***interrupt this sequence and move to another part of the program***.

Jump statements provide this capability by enabling the program to:

- *Exit a loop before it finishes*
- *Skip certain iterations of a loop*
- *Jump to a specific labeled statement*
- *Exit a function and return a value to the calling function*

These statements are particularly useful in loops, conditional blocks, and functions where the program logic requires sudden changes in the flow of execution.

The jump statements available in C include:

1. ***break statement***
2. ***continue statement***
3. ***goto statement***
4. ***return statement***

Each of these statements modifies program execution in a different way.

2.3.1. break Statement

The ***break statement*** *is used to immediately terminate the execution of a loop or switch statement. When a break statement is encountered, the program exits the loop or switch block and continues execution from the* ***first statement following that block***.

The break statement is commonly used when a specific condition occurs and the program no longer needs to continue the loop.

Situations where break is used

- *Exiting a loop when a required condition is met*
- *Preventing fall-through behavior in switch statements*
- *Stopping infinite loops*

Syntax

break;

Working of break in Loops

When a break statement appears inside a loop:

1. *The loop condition is evaluated.*
2. *The loop begins execution.*
3. *When break is encountered, the loop terminates immediately.*
4. *Control moves to the statement following the loop.*

Example

```
#include <stdio.h>

int main()
{
   int i;

   for(i = 1; i <= 10; i++)
   {
      if(i == 6)
      {
         break;
      }

      printf("%d ", i);
   }

   printf("\nLoop terminated");

   return 0;
}
```

Output

```
1 2 3 4 5
Loop terminated
```

Explanation

*The loop begins printing numbers from 1 to 10. When the value of i becomes **6**, the break statement executes and the loop stops immediately.*

2.3.2. continue Statement

*The **continue statement** is used to skip the remaining statements in the current iteration of a loop and move directly to the next iteration of the loop.*

*Unlike the break statement, continue **does not terminate the loop**. Instead, it only skips the remaining part of the current iteration.*

Syntax

```
continue;
```

Working of continue in Loops

When continue is encountered:

1. *Remaining statements in the loop body are skipped.*
2. *Control returns to the loop condition.*
3. *The next iteration of the loop begins.*

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
```

Explanation

*When i becomes **3**, the continue statement skips that iteration, so the number 3 is not printed.*

Difference Between break and continue

Aspect	***break Statement***	***continue Statement***
Definition	*The break statement is a jump control statement used to immediately terminate a loop or switch statement.*	*The continue statement is a jump control statement used to skip the remaining statements of the current loop iteration and move to the next iteration.*
Type of Statement	*Jump (control transfer) statement*	*Jump (control transfer) statement*
Main Purpose	*To stop the execution of a loop or switch statement completely.*	*To ignore the current iteration of a loop and proceed with the next iteration.*
Effect on Loop Execution	*The loop terminates immediately, and control transfers to the first statement after the loop.*	*The loop does not terminate; instead, it continues with the next iteration.*
Impact on Iterations	*All remaining iterations of the loop are cancelled once break is executed.*	*Only the current iteration is skipped; the loop continues executing the remaining iterations.*
Usage in Loops	*Can be used inside for, while, and do–while loops to terminate them early.*	*Can be used inside for, while, and do–while loops to skip specific iterations.*
Usage in switch Statement	*Commonly used to exit a switch block after a case is executed.*	*Cannot be used inside a switch statement independently (unless inside a loop).*
Control Flow	*Transfers control outside the loop or switch structure immediately.*	*Transfers control to the next iteration of the loop by skipping the remaining statements in the loop body.*
Behavior in for Loop	*Immediately exits the entire loop regardless of the remaining iterations.*	*Skips the rest of the current loop body and jumps to the **increment/decrement expression**, then continues with the next iteration.*
Behavior in while Loop	*Immediately exits the loop and moves control to the next statement after the loop.*	*Skips the remaining loop body and transfers control back to the **condition check** of the loop.*
Behavior in do–while Loop	*Terminates the loop immediately without checking the condition again.*	*Skips the remaining statements and moves to the **condition evaluation** step of the loop.*
Typical Programming Use	*Used when a specific condition is met and further execution of the loop is unnecessary, such as*	*Used when certain iterations must be ignored, such as skipping invalid inputs or unwanted values during processing.*

	when a required value is found in a list.	
Effect on Program Logic	*Causes early termination of the loop, which may improve efficiency when the desired condition is reached.*	*Maintains loop execution while avoiding unnecessary processing for certain iterations.*
Risk if Misused	*May cause premature termination of loops, which could lead to incomplete processing of data.*	*May cause infinite loops if the loop update statement is skipped or not executed properly.*
Program Efficiency	*Improves efficiency by avoiding unnecessary iterations after a condition is satisfied.*	*Improves efficiency by skipping unnecessary operations while continuing the loop execution.*

2.3.3. goto Statement

*The **goto statement** is used to transfer program control to a labeled statement within the same function. It provides an **unconditional jump** in the program.*

A label is an identifier followed by a colon (:) that marks a specific location in the program.

*Although goto can simplify certain complex control structures, excessive use of goto can make programs difficult to understand and maintain. Such programs are sometimes referred to as **spaghetti code**.*

Syntax

```
goto label;

/* statements */

label:
   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

Explanation

*The program jumps repeatedly to the label **start** until the condition i <= 5 becomes false.*

Uses of goto

- *Exiting multiple nested loops*
- *Error handling in system-level programming*
- *Jumping to cleanup sections in programs*

However, structured control statements like loops and conditionals are usually preferred over goto.

4. return Statement

*The **return statement** is used to terminate the execution of a function and return control back to the calling function. It may also return a value to the caller.*

*Every C program begins execution from the **main() function**, and when the return statement executes in main(), the program terminates.*

Syntax

return value;

or

return;

Example

```
#include <stdio.h>

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

   result = add(4, 6);

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

   return 0;
}
```

Explanation

- *The function add() calculates the sum of two numbers.*
- *The result is returned to the main() function using return.*
- *The returned value is stored in the variable result.*

3.1 Functions

*A **function** in C programming is a self-contained block of statements designed to perform a specific task. Functions are one of the most important concepts in structured programming because they allow programmers to divide a large and complex program into smaller and manageable modules. Each module performs a particular operation and can be executed whenever required by calling the function.*

In many programming situations, certain operations must be performed repeatedly. Instead of writing the same sequence of instructions multiple times, a function allows the programmer to define the operation once and reuse it whenever necessary. This approach reduces code duplication, improves program readability, and simplifies program maintenance.

*Functions support the principle of **modular programming**, where a complex problem is divided into smaller parts, each handled by a separate function. Each function focuses on a specific task, which makes debugging, testing, and modification easier.*

*For example, in a program designed to perform mathematical calculations, separate functions can be written for **addition, subtraction, multiplication, and division**. These functions can then be called whenever the respective operations are required.*

Functions in C can be classified into two major categories:

- ***Library Functions***
- ***User-Defined Functions***

Library functions *are predefined functions provided by the C standard library, such as printf(), scanf(), sqrt(), and strlen(). These functions are already implemented and can be used directly in a program.*

User-defined functions *are functions created by programmers to perform specific tasks according to the requirements of a program.*

3.1.1 Definition and Declaration of Functions

A function in C typically involves two important aspects: ***function declaration*** *and* ***function definition****.*

Function Definition

The ***function definition*** *is the actual implementation of a function. It describes what the function does when it is executed. The function definition includes the function name, return type, parameters, and the body of the function that contains the executable statements.*

The general structure of a function definition consists of the following components:

- ***Return type*** *– specifies the type of value returned by the function.*
- ***Function name*** *– the identifier used to call the function.*
- ***Parameter list*** *– variables used to receive input values.*
- ***Function body*** *– contains the statements that perform the required task.*

Syntax

```
return_type function_name(parameter_list)
{
   // statements
}
```

Example

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

In this example:

- *int is the return type.*
- *add is the function name.*
- *a and b are parameters.*
- *The function calculates and returns the sum.*

Classification of Functions in C

In the C programming language, functions can be broadly classified into two major categories based on their origin and purpose. These categories are ***Library Functions*** *and* ***User-Defined Functions****. Both types of functions help in simplifying programming tasks and improving code organization.*

Built-in Functions/1. Library Functions

Built-in functions are ***predefined functions provided by the C programming language through its standard libraries****. These functions are already implemented and optimized by the developers of the language. Therefore, programmers can directly use them in their programs instead of writing the entire logic from scratch.*

In programming, many tasks such as ***mathematical calculations, string manipulation, character classification, and time operations*** *are frequently required. Writing separate code for each of these operations every time would make programs long, complex, and difficult to maintain. Built-in functions solve this problem by providing* ***ready-made solutions for commonly used operations****.*

The use of built-in functions provides several advantages:

- ***Reduces program length and complexity***
- ***Improves code readability and maintainability***
- ***Saves development time***
- ***Ensures reliability since functions are pre-tested***
- ***Provides efficient implementation for common operations***

In C programming, built-in functions are organized into ***standard library header files****. To use a particular function, the corresponding header file must be included in the program using the* ***#include directive****.*

Example:

```
#include <math.h>
#include <string.h>
#include <ctype.h>
#include <time.h>
```

Each header file contains a group of functions designed to perform a particular category of operations.

Classification of Built-in Functions

Built-in functions in C are broadly classified into the following categories:

1. ***Mathematical Functions***
2. ***String Functions***
3. ***Character Functions***
4. ***Date and Time Functions***

Each category performs specific types of operations related to data processing.

3.2.1 Mathematical Functions

*Mathematical functions are built-in functions used to perform **various mathematical and scientific calculations**. These functions are particularly useful in programs involving **engineering computations, statistical calculations, scientific simulations, and numeric processing**.*

*Mathematical functions are defined in the **math.h** header file.*

Header File

#include <math.h>

*These functions perform operations such as **square roots, exponentiation, logarithmic calculations, trigonometric functions, and rounding operations**.*

Common Mathematical Functions

Function	***Description***	***Example***
sqrt(x)	*Returns square root of x*	*sqrt(25) → 5*
pow(x,y)	*Returns x raised to the power y*	*pow(2,3) → 8*
ceil(x)	*Rounds value upward to nearest integer*	*ceil(4.3) → 5*
floor(x)	*Rounds value downward to nearest integer*	*floor(4.8) → 4*
abs(x)	*Returns absolute value*	*abs(-5) → 5*
log(x)	*Returns natural logarithm of x*	*log(10)*
sin(x)	*Calculates sine value*	*sin(angle)*
cos(x)	*Calculates cosine value*	*cos(angle)*

Example Program

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

int main()
{
   double num = 16;

   printf("Square root: %.2f\n", sqrt(num));
   printf("Power: %.2f\n", pow(2,3));

   return 0;
}
```

These functions allow programmers to perform complex mathematical operations with ***minimal coding effort****.*

3.2.2 String Functions

String functions are used to perform operations on ***strings****, which are sequences of characters stored in arrays of type char. Handling strings manually can be complicated, so the C language provides built-in functions to simplify string manipulation.*

String functions are defined in the ***string.h*** *header file.*

Header File

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

These functions perform operations such as ***copying strings, concatenating strings, comparing strings, and calculating string length****.*

Common String Functions

Function	***Description***	***Syntax***	***Example***	***Output***
strlen()	*Finds the length of a string*	*strlen(string);*	*c char str[] = "Hello"; printf("%d", strlen(str));*	*5*
strcpy()	*Copies one string into another*	*strcpy(destination, source);*	*c char s1[20]; char s2[] = "Hello"; strcpy(s1, s2); printf("%s", s1);*	*Hello*
strcat()	*Concatenates two strings*	*strcat(destination, source);*	*c char s1[20] = "Hello "; char s2[] = "World"; strcat(s1, s2); printf("%s",*	*Hello World*

			s1);	
strcmp()	Compares two strings	strcmp(str1, str2);	c char s1[] = "Apple"; char s2[] = "Apple"; printf("%d", strcmp(s1, s2));	0
strncpy()	Copies specified number of characters	strncpy(destination, source, n);	c char s1[20]; char s2[] = "Programming"; strncpy(s1, s2, 4); s1[4] = '\0'; printf("%s", s1);	Prog
strstr()	Finds substring in a string	strstr(main_string, sub_string);	c char str[] = "Hello World"; char *p = strstr(str, "World"); printf("%s", p);	World

Example Program

```
#include <stdio.h>
#include <string.h>
int main()
{
   char str1[20] = "Hello";
   char str2[20] = "C";
   strcat(str1, str2);
   printf("Result: %s", str1);
   return 0;
}
```

*String functions are widely used in **text processing, file handling, and database applications**.*

3.2.3 Character Functions

*Character functions are used to test and manipulate **individual characters**. These functions are useful when programs need to check the type of character entered by the user.*

*Character functions are defined in the **ctype.h** header file.*

Header File

```
#include <ctype.h>
```

*These functions help determine whether a character is **alphabetic, numeric, uppercase, lowercase, or special symbol**.*

Common Character Functions

Function	Description	Syntax	Example	Output
isalpha()	Checks whether a character is an alphabet	isalpha(character);	c char ch = 'A'; if(isalpha(ch)) printf("Alphabet");	Alphabet
isdigit()	Checks whether a character is a digit	isdigit(character);	c char ch = '5'; if(isdigit(ch)) printf("Digit");	Digit
isalnum()	Checks whether a character is alphanumeric (letter or digit)	isalnum(character);	c char ch = '9'; if(isalnum(ch)) printf("Alphanumeric");	Alphanumeric
islower()	Checks whether a character is lowercase	islower(character);	c char ch = 'a'; if(islower(ch)) printf("Lowercase");	Lowercase
isupper()	Checks whether a character is uppercase	isupper(character);	c char ch = 'A'; if(isupper(ch)) printf("Uppercase");	Uppercase
tolower()	Converts uppercase character to lowercase	tolower(character);	c char ch = 'A'; printf("%c", tolower(ch));	a
toupper()	Converts lowercase character to uppercase	toupper(character);	c char ch = 'b'; printf("%c", toupper(ch));	B

Example Program

```
#include <stdio.h>
#include <ctype.h>
int main()
{
   char ch = 'a';
   if(islower(ch))
      printf("Lowercase character");
   return 0;
}
```

Character functions are especially useful in ***input validation, text processing, and character classification programs****.*

3.2.4 Date and Time Functions

Date and time functions are used to obtain and manipulate the ***current system date and time****. These functions are helpful in programs that require time-based operations such as* ***event logging, scheduling, timers, and monitoring systems****.*

Date and time functions are defined in the ***time.h*** *header file.*

Header File

#include <time.h>

These functions allow programs to retrieve the current system time and convert it into readable formats.

Common Date and Time Functions

Function	*Description*	*Syntax*	*Example*	*Output*
time()	*Returns the current system time (in seconds since Jan 1, 1970)*	*time(&t);*	*c time_t t; time(&t); printf("%ld", t);*	*1710325000 (example value)*
ctime()	*Converts time value to a readable string format*	*ctime(&t);*	*c time_t t; time(&t); printf("%s", ctime(&t));*	*Wed Mar 13 12:30:45 2026*
localtime()	*Converts time value into local date and time structure*	*localtime(&t);*	*c time_t t; struct tm *info; time(&t); info = localtime(&t); printf("%d", info->tm_year + 1900);*	*2026*
asctime()	*Converts tm structure to readable string*	*asctime(time_structure);*	*c time_t t; struct tm *info; time(&t); info = localtime(&t); printf("%s", asctime(info));*	*Wed Mar 13 12:30:45 2026*
difftime()	*Calculates difference*	*difftime(time1, time2);*	*c time_t start, end; start = time(NULL);*	*0.000000*

	between two times		end = time(NULL); printf("%f", difftime(end, start));	

Example Program

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

int main()
{
   time_t t;
   time(&t);

   printf("Current time: %s", ctime(&t));

   return 0;
}
```

*This program retrieves the **current system time and displays it in readable form**.*

3.User-Defined Functions

*User-defined functions are functions that are **created and defined by the programmer** to perform specific tasks according to the requirements of a program. Unlike library functions, which are provided by the C standard library, user-defined functions are written by the programmer to solve particular problems within the program.*

*Large programs can become difficult to understand, manage, and debug if all the statements are written in a single block. User-defined functions help overcome this problem by **dividing a program into smaller, manageable modules**. Each function performs a specific task, making the program more organized and structured.*

3.1.1 User-defined functions provide several advantages:

- ***Modularity:** Divides a program into smaller logical parts*
- ***Code Reusability:** The same function can be used multiple times*
- ***Improved Readability:** Programs become easier to understand*
- ***Easier Debugging:** Errors can be located and corrected more easily*
- ***Better Program Maintenance:** Changes can be made without affecting the entire program*

In C programming, functions are designed to perform specific tasks and communicate with the calling program through ***arguments and return values****.*

3.1.2 Elements of Functions

A function in C consists of several important elements that define how the function is declared, implemented, and executed within a program. These elements help the compiler understand the structure of the function and allow the program to communicate with the function correctly.

The main elements of a function include:

1. ***Function Declaration (Prototype)***
2. ***Function Definition***
3. ***Return Statement***

Function Declaration

A ***function declaration*** *informs the compiler about the existence of a function before it is used in the program. It specifies the function name, return type, and parameter types without providing the function body.*

Function declarations are important because they allow the compiler to perform ***type checking*** *and ensure that the function is called correctly.*

Example

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

This statement tells the compiler that there exists a function named add which takes two integer arguments and returns an integer value.

Function Prototype

A ***function prototype*** *is another name for a function declaration. It defines the function's interface by specifying the return type, function name, and parameter types.*

The main purpose of a function prototype is to ensure that the function is called correctly. If the arguments passed to the function do not match the prototype, the compiler can generate an error during compilation.

Syntax

```
return_type function_name(type1, type2, ...);
```

Example

```
int multiply(int, int);
```

Function prototypes are usually placed at the beginning of the program, before the main() function.

Return Statement

*The **return statement** is used to terminate the execution of a function and return a value to the calling function. When the return statement is executed, control immediately transfers back to the point from which the function was called.*

The value specified in the return statement must match the return type of the function.

Syntax

return expression;

Example

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

In this example, the function calculates the square of a number and returns the result.

*A function may also be declared with the return type **void**, which indicates that it does not return any value.*

Example

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

In this case, the function performs an action but does not return a value.

Communication Between Functions

When functions interact with other parts of a program, two important elements are involved:

1. Arguments (Parameters)

*Arguments are the values that are **passed to a function when it is called**. These values provide input data that the function uses to perform its operations.*

2. Return Value

*The return value is the **result produced by the function after completing its task**. This result is returned to the calling function using the return statement.*

Depending on whether arguments and return values are used or not, user-defined functions are classified into four main types.

1. Function with No Arguments and No Return Value

*In this type of function, the function **does not receive any input from the calling function** and **does not return any value** after execution. The function simply performs a particular task when it is called.*

*Since there is no value returned, the return type of such functions is usually **void**. These functions are commonly used when the program needs to perform actions such as **displaying messages, printing results, or performing operations that do not require input from other parts of the program**.*

Conceptual Working

1. *The function is defined without parameters.*
2. *The function performs the required task internally.*
3. *The function completes execution and control returns to the calling function.*
4. *No value is returned.*

Example

```
#include <stdio.h>

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

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

Explanation

The function display() does not accept any parameters and does not return any value. It simply prints a message when called.

2. Function with Arguments but No Return Value

In this type of function, the function ***accepts input values from the calling function but does not return any result****. The arguments provided by the caller are used within the function to perform calculations or operations.*

The function may display the result directly instead of returning it to the calling function.

These functions are useful when the program needs to ***process given data and immediately display the result without passing the result back****.*

Conceptual Working

1. *The calling function sends arguments to the function.*
2. *The function receives these values through parameters.*
3. *The function processes the values.*
4. *The result may be displayed but is not returned.*

Example

```
#include <stdio.h>

void add(int a, int b)
{
   int sum = a + b;
   printf("Sum = %d\n", sum);
}

int main()
{
   add(5, 7);
   return 0;
}
```

Explanation

The function add() receives two values as parameters, calculates their sum, and prints the result.

3. Function with Arguments and Return Value

*This is the **most commonly used type of function** in C programming. In this type, the function receives arguments from the calling function, processes the data, and returns the computed result back to the calling function.*

*This type of function enables **two-way communication** between the function and the calling program.*

Conceptual Working

1. *The calling function passes arguments.*
2. *The function receives the arguments.*
3. *The function processes the data.*
4. *The result is returned using the return statement.*
5. *The calling function receives the returned value.*

Example

```
#include <stdio.h>

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

int main()
{
   int result;

   result = add(10, 20);

   printf("Sum = %d\n", result);

   return 0;
}
```

Explanation

The function add() receives two numbers, calculates their sum, and returns the result to the main() function.

4. Function with No Arguments but Return Value

In this type of function, the function ***does not accept any arguments from the calling function but returns a value*** *after performing its operation.*

The function internally generates or obtains the required value, processes it, and returns the result.

This type is useful when the function itself is responsible for ***generating the input value****, such as reading input from the user.*

Conceptual Working

1. *The function does not receive arguments.*
2. *The function generates or reads input internally.*
3. *The function processes the data.*
4. *The result is returned to the calling function.*

Example

```
#include <stdio.h>

int getNumber()
{
   int num;
   printf("Enter a number: ");
   scanf("%d", &num);
   return num;
}

int main()
{
   int value;

   value = getNumber();

   printf("Entered number = %d\n", value);

   return 0;
}
```

Explanation

The function getNumber() reads a number from the user and returns the value to the main() function.

5.7.3 Class

*In object-oriented programming, a **class** is one of the most fundamental concepts. A class is a **user-defined data type** that acts as a blueprint or template for creating objects. It defines the structure and behavior of objects by specifying the **data members (variables)** and **member functions (functions)** that the objects will have.*

*A class does not represent an actual object; instead, it describes the properties and behaviors that objects created from the class will possess. Therefore, a class can be considered as a **logical representation**, while objects are the **physical instances** of that representation.*

*For example, consider the concept of a **student**. A class named Student can be created to define attributes such as roll number, name, and marks, along with functions to display or modify these details. Once the class is defined, multiple student objects can be created using the same class.*

*Thus, a **class provides a way to group data and functions together into a single unit**, which supports the principle of **encapsulation** in object-oriented programming.*

*A **class** is a user-defined data type that contains **data members and member functions**, which together define the properties and behaviors of objects.*

Syntax of Class

The general syntax for defining a class in C++ is:

```
class ClassName
{
   access_specifier:
      data_members;

      member_functions();
};
```

Explanation

- ***class** – keyword used to define a class*
- ***ClassName** – name of the class*
- ***data members** – variables that store object data*
- ***member functions** – functions that operate on the data members*
- ***access specifiers** – control the accessibility of data members and member functions*

Example of Class

```
class Student
{
   int rollno;
```

```
    char name[20];

public:
    void display();
};
```

In this example:

- *Student is the class name.*
- *rollno and name are data members.*
- *display() is a member function.*

Characteristics of a Class

- *A class is a **blueprint for creating objects**.*
- *It groups **data and functions into a single unit**.*
- *It supports **data encapsulation** and **data abstraction**.*
- *A class does not occupy memory until **objects are created**.*
- *Multiple objects can be created from a single class.*

5.7.4 Object

*An **object** is an instance of a class. While a class defines the structure and behavior of entities, objects represent the **actual entities that exist during program execution**. Objects contain **real values for the data members defined in the class** and can use the member functions defined in the class.*

*Each object created from a class has its **own copy of data members**, but all objects share the same member functions defined in the class.*

For example, if Student is a class, then s1, s2, and s3 can be objects of that class representing different students.

*Objects are the fundamental components of object-oriented programming because they allow programs to model **real-world entities and interactions**.*

*An **object** is an instance of a class that contains actual values for the data members and can access the member functions of the class.*

Syntax for Creating an Object

The general syntax for creating an object is:

ClassName objectName;

Example

Student s1;

Here:

- *Student is the class name.*
- *s1 is an object of the class Student.*

Accessing Members Using Objects

*Members of a class can be accessed using the **dot operator (.)**.*

Example:

s1.display();

In this statement:

- *s1 is the object.*
- *display() is the member function being called.*

Example Program

```
#include <iostream>
using namespace std;

class Student
{
public:
   int rollno;
   float marks;

   void display()
   {
      cout << "Roll Number: " << rollno << endl;
      cout << "Marks: " << marks << endl;
   }
};

int main()
{
   Student s1;

   s1.rollno = 101;
   s1.marks = 95.5;

   s1.display();

   return 0;
}
```

Characteristics of Objects

- *Objects are **instances of classes**.*
- *Each object has its **own set of data members**.*
- *Objects can **access member functions of the class**.*
- *Objects represent **real-world entities in a program**.*
- *Multiple objects can be created from the same class.*

5.7.5 Inheritance

Definition

*Inheritance is a mechanism in C++ by which **one class acquires the properties and behaviors of another class**. It allows a new class to reuse the **data members and member functions** of an existing class.*

*The class whose properties are inherited is called the **base class (or parent class)**, and the class that inherits these properties is called the **derived class (or child class)**.*

*Inheritance helps in **code reuse, reducing redundancy, and building hierarchical relationships between classes**.*

Syntax of Inheritance

The general syntax for inheritance in C++ is:

```
class DerivedClass : access_specifier BaseClass
{
   // members of derived class
};
```

Explanation

- ***DerivedClass** – The class that inherits properties.*
- ***BaseClass** – The class whose properties are inherited.*
- ***access_specifier** – Specifies the type of inheritance such as public, private, or protected.*

Example

```
#include <iostream>
using namespace std;

class Person
```

```
{
public:
   void display()
   {
      cout << "This is the base class" << endl;
   }
};

class Student : public Person
{
};

int main()
{
   Student s1;
   s1.display();
   return 0;
}
```

Explanation

- *Person is the **base class**.*
- *Student is the **derived class**.*
- *The object s1 of the Student class can access the function display() because it is inherited from the Person class.*

Types of Inheritance

C++ supports the following types of inheritance:

1. ***Single Inheritance** – One derived class inherits from one base class.*
2. ***Multiple Inheritance** – One derived class inherits from more than one base class.*
3. ***Multilevel Inheritance** – A class inherits from another derived class.*
4. ***Hierarchical Inheritance** – Multiple derived classes inherit from a single base class.*
5. ***Hybrid Inheritance** – Combination of two or more types of inheritance.*

Types of Inheritance

*C++ supports several types of inheritance depending on how classes are related to each other. These types describe the different ways in which **derived classes inherit the properties and behaviors of base classes**. The major types of inheritance are explained below.*

1. Single Inheritance

Single inheritance is the simplest form of inheritance. In this type, ***one derived class inherits the properties and functions of a single base class****. The derived class can access the features of the base class and may also add its own additional members.*

Example:
If a class ***Student*** *inherits from a class* ***Person****, then Student automatically gains the attributes and functions of the Person class.*

2. Multiple Inheritance

In multiple inheritance, ***a single derived class inherits properties from more than one base class****. This allows the derived class to combine the features of multiple classes into one.*

Example:
A class ***TeachingAssistant*** *may inherit from both* ***Student*** *and* ***Teacher*** *classes. This means the TeachingAssistant class can use features of both parent classes.*

3. Multilevel Inheritance

In multilevel inheritance, ***a class is derived from another derived class****, forming a chain of inheritance. This creates a hierarchical relationship among classes.*

Example:
A class ***Person*** *may be the base class. A class* ***Student*** *may inherit from Person, and another class* ***GraduateStudent*** *may inherit from Student.*

Thus, the inheritance chain becomes:
Person → Student → GraduateStudent

4. Hierarchical Inheritance

In hierarchical inheritance, ***multiple derived classes inherit from the same base class****. Each derived class can use the features of the base class while also defining its own additional members.*

Example:
A class ***Person*** *may be inherited by classes such as* ***Student****,* ***Teacher****, and* ***Employee****. All these derived classes share the common features of the Person class.*

5. Hybrid Inheritance

Hybrid inheritance is a ***combination of two or more types of inheritance****. It occurs when different inheritance structures are used together in a program.*

For example, a program may combine ***multilevel inheritance and multiple inheritance*** *to form a complex class hierarchy.*

5.7.6 Polymorphism in C++

Polymorphism is one of the core principles of object-oriented programming in C++. The word ***polymorphism*** *comes from the Greek words poly meaning* ***many*** *and morph meaning* ***forms****. Therefore, polymorphism refers to the ability of a* ***single interface, function, or operator to behave in multiple ways depending on the situation****.*

In C++, polymorphism allows the same function name, operator, or object to perform different operations based on the type of data or the object that calls it. This feature makes programs more flexible and reduces the need to write separate functions for similar tasks.

Polymorphism

The concept of polymorphism is based on the idea that ***a single operation can take many forms****. In object-oriented programming, the same function name can be used for different purposes, depending on the context in which it is used.*

For example, consider a function named ***calculate()****. This function might calculate the result for different operations such as addition, multiplication, or finding the area of shapes. Although the function name remains the same, the actual operation performed may vary depending on the parameters or the type of object used.*

This approach improves program design because it allows programmers to use a ***common interface for different implementations****.*

Role of Polymorphism in Object-Oriented Programming

Polymorphism plays a key role in building ***flexible and reusable software systems****. It allows different objects to respond to the same function call in their own way. Because of this, programmers can write programs that are easier to extend and modify without changing existing code.*

Polymorphism also helps in creating programs that more closely resemble ***real-world systems****, where similar actions can produce different results depending on the object performing the action.*

For example, the action ***draw()*** *may produce different shapes depending on whether the object represents a circle, rectangle, or triangle.*

Types of Polymorphism in C++

In C++, polymorphism is mainly categorized into two types based on when the function call is resolved.

1. Compile-Time Polymorphism (Static Polymorphism)

Compile-time polymorphism occurs when the decision about which function to execute is made ***during program compilation****. The compiler determines the appropriate function based on the number and type of arguments provided.*

This type of polymorphism is usually achieved through:

- ***Function Overloading***
- ***Operator Overloading***

In function overloading, multiple functions can have the same name but different parameter lists. The compiler identifies the correct function to call by examining the arguments passed.

*Operator overloading allows standard operators such as +, -, or * to work with user-defined data types, enabling them to perform operations beyond their default behavior.*

Example of function overloading:

```
#include <iostream>
using namespace std;

class Addition
{
public:
   int add(int a, int b)
   {
      return a + b;
   }

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

```
int main()
{
   Addition obj;
   cout << obj.add(5, 3) << endl;
   cout << obj.add(2.5f, 3.5f) << endl;

   return 0;
}
```

In this example, the function add() performs different operations depending on the type of arguments.

2. Run-Time Polymorphism (Dynamic Polymorphism)

Run-time polymorphism occurs when the decision about which function to execute is made ***during program execution*** *rather than at compile time.*

This type of polymorphism is achieved through ***function overriding using virtual functions****. In this case, a derived class provides its own version of a function that is already defined in the base class.*

When a function is called through a base class reference or pointer, the program determines at runtime which version of the function should be executed. This allows programs to behave dynamically depending on the type of object being used.

```
#include <iostream>
using namespace std;
class Base
{
public:
   virtual void show()
   {
      cout << "This is base class function" << endl;
   }
};

class Derived : public Base
{
public:
   void show()
   {
      cout << "This is derived class function" << endl;
```

```
    }
};

int main()
{
    Base *b;
    Derived d;

    b = &d;
    b->show();

    return 0;
}
```

5.7.7 Encapsulation

Definition

Encapsulation is a fundamental concept of object-oriented programming in C++. It refers to the process of ***combining data (variables) and functions (methods) that operate on that data into a single unit called a class****. In addition to grouping data and functions together, encapsulation also involves* ***restricting direct access to certain parts of the data*** *to protect it from unauthorized modification.*

Thus, encapsulation provides ***data protection and controlled access*** *to the data members of a class.*

Encapsulation

The main idea behind encapsulation is ***data hiding and data protection****. In object-oriented programming, the internal details of a class are hidden from the outside world, and the data can be accessed only through specific functions defined within the class.*

This means that users of a class cannot directly modify its data members unless the class allows it through ***public member functions****. By doing so, encapsulation helps maintain the* ***integrity and security of data****.*

Encapsulation is achieved in C++ using ***classes and access specifiers*** *such as:*

- ***Private***
- ***Protected***
- ***Public***

These access specifiers control how data members and member functions can be accessed.

Access Specifiers in Encapsulation

Access specifiers play an important role in implementing encapsulation.

Private

*When data members are declared as **private**, they cannot be accessed directly from outside the class. They can only be accessed through the member functions of the class.*

Public

*Members declared as **public** can be accessed from anywhere in the program.*

Protected

*Members declared as **protected** can be accessed within the class and by derived classes through inheritance.*

Example of Encapsulation

```
#include <iostream>
using namespace std;

class Student
{
private:
   int rollno;

public:
   void setRollNo(int r)
   {
      rollno = r;
   }

   void display()
   {
      cout << "Roll Number: " << rollno;
   }
};

int main()
{
   Student s1;
```

```
    s1.setRollNo(101);
    s1.display();

    return 0;
}
```

Explanation

In this example:

- *rollno is declared as a **private data member**.*
- *It cannot be accessed directly outside the class.*
- *The functions setRollNo() and display() are **public member functions** used to modify and display the data.*

*Thus, the data is **protected and accessed only through controlled methods**.*

5.7.8 Abstraction

Abstraction

*Abstraction is a fundamental concept of object-oriented programming in C++. It refers to the process of **hiding the internal implementation details and showing only the essential features of an object to the user**. In other words, abstraction focuses on **what an object does rather than how it performs the task**.*

*The main purpose of abstraction is to **simplify complex systems** by hiding unnecessary details and allowing users to interact only with the important parts of a program.*

*For example, when a person uses a **mobile phone**, they simply press buttons to make calls or send messages. The internal processes such as signal transmission, network communication, and hardware operations remain hidden from the user. This is an example of abstraction in real life.*

Similarly, in programming, abstraction hides complex implementation details and provides a simple interface for users.

Example of Abstraction in C++

```
#include <iostream>
using namespace std;

class Calculator
{
public:
    int add(int a, int b)
    {
```

```
        return a + b;
    }
};

int main()
{
    Calculator c1;
    cout << "Sum = " << c1.add(5, 3);
    return 0;
}
```

Explanation

In this example:

- *Calculator is a class that performs addition.*
- *The function add() is used to add two numbers.*
- *The user simply calls c1.add(5,3) to get the result.*

*The user does not need to know **how the addition operation is implemented internally**. They only need to know how to use the function. The internal working of the function remains hidden inside the class.*

*This demonstrates the concept of **abstraction**, where only the necessary functionality is visible to the user.*