Lenora College Of Engineering

Rampachodavaram

B.Tech – CSE – R23

Introduction to Programming - C

L	T	P	C
3	0	0	3

INTRODUCTION TO PROGRAMMING

(Common to All branches of Engineering)

Course Objectives:

- To introduce students to the fundamentals of computer programming.
- To provide hands-on experience with coding and debugging.
- To foster logical thinking and problem-solving skills using programming.
- To familiarize students with programming concepts such as data types, control structures, functions, and arrays.
- To encourage collaborative learning and teamwork in coding projects.

Course Outcomes: A student after completion of the course will be able to

CO1: Understand basics of computers, the concept of algorithm and algorithmic thinking.
CO2: Analyse a problem and develop an algorithm to solve it.
CO3: Implement various algorithms using the C programming language.
CO4: Understand more advanced features of C language.
CO5: Develop problem-solving skills and the ability to debug and optimize the code.

UNIT I Introduction to Programming and Problem Solving

History of Computers, Basic organization of a computer: ALU, input-output units, memory, program counter, Introduction to Programming Languages, Basics of a Computer Program-Algorithms, flowcharts (Using Dia Tool), pseudo code. Introduction to Compilation and Execution, Primitive Data Types, Variables, and Constants, Basic Input and Output, Operations, Type Conversion, and Casting.

Problem solving techniques: Algorithmic approach, characteristics of algorithm, Problem solving strategies: Top-down approach, Bottom-up approach, Time and space complexities of algorithms.

UNIT II Control Structures

Simple sequential programs Conditional Statements (if, if-else, switch), Loops (for, while, do-while) Break and Continue.

UNIT III Arrays and Strings

Arrays indexing, memory model, programs with array of integers, two dimensional arrays, Introduction to Strings.

UNIT IV Pointers & User Defined Data types

Pointers, dereferencing and address operators, pointer and address arithmetic, array manipulation using pointers, User-defined data types-Structures and Unions.

UNIT V Functions & File Handling

Introduction to Functions, Function Declaration and Definition, Function call Return Types and Arguments, modifying parameters inside functions using pointers, arrays as parameters. Scope and Lifetime of Variables, Basics of File Handling

UNIT I

Introduction to Programming and Problem Solving: History of Computers, Basic organization of a computer: ALU, input-output units, memory, program counter, Introduction to Programming Languages, Basics of a Computer Program- Algorithms, flowcharts (Using Dia Tool), pseudo code. Introduction to Compilation and Execution, Primitive Data Types, Variables, and Constants, Basic Input and Output, Operations, Type Conversion, and Casting.

Introduction to Programming Languages

Introduction:

A programming language is a set of instructions and syntax used to create software programs. Some of the key features of programming languages include:

1. **Syntax**: The specific rules and structure used to write code in a programming language.
2. **Data Types**: The type of values that can be stored in a program, such as numbers, strings, and booleans.
3. **Variables**: Named memory locations that can store values.
4. **Operators**: Symbols used to perform operations on values, such as addition, subtraction, and comparison.
5. **Control Structures**: Statements used to control the flow of a program, such as if-else statements, loops, and function calls.
6. **Libraries** and Frameworks: Collections of pre-written code that can be used to perform common tasks and speed up development.
7. **Paradigms**: The programming style or philosophy used in the language, such as procedural, object-oriented, or functional.

Examples of popular programming languages include Python, Java, C++, JavaScript, and Ruby. Each language has its own strengths and weaknesses and is suited for different types of projects.

A programming language is a formal language that specifies a set of instructions for a computer to perform specific tasks. It's used to write software programs and applications, and to control and manipulate computer systems. There are many different programming languages, each with its own syntax, structure, and set of commands. Some of the most commonly used programming languages include Java, Python, C++, JavaScript, and C#. The choice of programming language depends on the specific requirements of a project, including the platform being used, the intended audience, and the desired outcome. Programming languages continue to evolve and change over time, with new languages being developed and older ones being updated to meet changing needs.

Are you aiming to become a software engineer one day? Do you also want to develop a mobile application that people all over the world would love to use? Are you passionate enough to take

the big step to enter the world of programming? Then you are in the right place because through this article you will get a brief introduction to programming. Now before we understand what programming is, you must know what is a computer. A computer is a device that can accept human instruction, processes it, and responds to it or a computer is a computational device that is used to process the data under the control of a computer program. Program is a sequence of instruction along with data.

The basic components of a computer are:

1. Input unit
2. Central Processing Unit(CPU)
3. Output unit

The CPU is further divided into three parts-

- Memory unit
- Control unit
- Arithmetic Logic unit

Most of us have heard that CPU is called the brain of our computer because it accepts data, provides temporary memory space to it until it is stored(saved) on the hard disk, performs logical operations on it and hence processes(here also means converts) data into information. We all know that a computer consists of hardware and software. Software is a set of programs that performs multiple tasks together. An operating system is also software (system software) that helps humans to interact with the computer system.
A program is a set of instructions given to a computer to perform a specific operation. or computer is a computational device that is used to process the data under the control of a computer program. While executing the program, raw data is processed into the desired output format. These computer programs are written in a programming language which are high-level languages. High level languages are nearly human languages that are more complex than the computer understandable language which are called machine language, or low level language. So after knowing the basics, we are ready to create a very simple and basic program. Like we have different languages to communicate with each other, likewise, we have different languages like C, C++, C#, Java, python, etc to communicate with the computers. The computer only understands binary language (the language of 0's and 1's) also called machine-understandable language or low-level language but the programs we are going to write are in a high-level language which is almost similar to human language.
The piece of code given below performs a basic task of printing "hello world! I am learning programming" on the console screen. We must know that keyboard, scanner, mouse, microphone, etc are various examples of input devices, and monitor(console screen), printer, speaker, etc are examples of output devices.

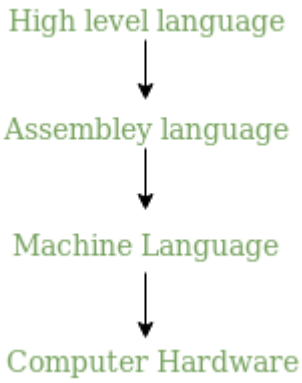
```
main()

{

 clrscr();

 printf("hello world! I am learning to program");
```

```
getch();
}
```

At this stage, you might not be able to understand in-depth how this code prints something on the screen. The main() is a standard function that you will always include in any program that you are going to create from now onwards. Note that the execution of the program starts from the main() function. The clrscr() function is used to see only the current output on the screen while the printf() function helps us to print the desired output on the screen. Also, getch() is a function that accepts any character input from the keyboard. In simple words, we need to press any key to continue(some people may say that getch() helps in holding the screen to see the output).

Between high-level language and machine language, there are assembly languages also called symbolic machine code. Assembly languages are particularly computer architecture specific. Utility program (**Assembler**) is used to convert assembly code into executable machine code. High Level Programming Language is portable but requires Interpretation or compiling to convert it into a machine language that is computer understood.

Hierarchy of Computer language -



There have been many programming languages some of them are listed below:

C	**Python**	**C++**
C#	**R**	**Ruby**
COBOL	**ADA**	**Java**
Fortran	**BASIC**	**Altair BASIC**
True BASIC	**Visual BASIC**	**GW BASIC**

QBASIC	**PureBASIC**	**PASCAL**
Turbo Pascal	**GO**	**ALGOL**
LISP	**SCALA**	**Swift**
Rust	**Prolog**	**Reia**
Racket	**Scheme**	**Simula**
Perl	**PHP**	**Java Script**
CoffeeScript	**VisualFoxPro**	**Babel**
Logo	**Lua**	**Smalltalk**
Matlab	**F**	**F#**
Dart	**Datalog**	**dbase**
Haskell	**dylan**	**Julia**
ksh	**metro**	**Mumps**
Nim	**OCaml**	**pick**
TCL	**D**	**CPL**
Curry	**ActionScript**	**Erlang**

Clojure	DarkBASCIC	Assembly

Most Popular Programming Languages -

- C
- Python
- C++
- Java
- SCALA
- C#
- R
- Ruby
- Go
- Swift
- JavaScript

Characteristics of a programming Language -

- A programming language must be simple, easy to learn and use, have good readability, and be human recognizable.
- Abstraction is a must-have Characteristics for a programming language in which the ability to define the complex structure and then its degree of usability comes.
- A portable programming language is always preferred.
- Programming language's efficiency must be high so that it can be easily converted into a machine code and its execution consumes little space in memory.
- A programming language should be well structured and documented so that it is suitable for application development.
- Necessary tools for the development, debugging, testing, maintenance of a program must be provided by a programming language.
- A programming language should provide a single environment known as Integrated Development Environment(IDE).
- A programming language must be consistent in terms of syntax and semantics.

Basic Terminologies in Programming Languages:

- **Algorithm**: A step-by-step procedure for solving a problem or performing a task.
- **Variable**: A named storage location in memory that holds a value or data.

- **Data Type**: A classification that specifies what type of data a variable can hold, such as integer, string, or boolean.
- **Function**: A self-contained block of code that performs a specific task and can be called from other parts of the program.
- **Control Flow**: The order in which statements are executed in a program, including loops and conditional statements.
- **Syntax**: The set of rules that govern the structure and format of a programming language.
- **Comment**: A piece of text in a program that is ignored by the compiler or interpreter, used to add notes or explanations to the code.
- **Debugging**: The process of finding and fixing errors or bugs in a program.
- **IDE**: Integrated Development Environment, a software application that provides a comprehensive development environment for coding, debugging, and testing.
- **Operator**: A symbol or keyword that represents an action or operation to be performed on one or more values or variables, such as + (addition), - (subtraction), * (multiplication), and / (division).
- **Statement**: A single line or instruction in a program that performs a specific action or operation.

Advantages of programming languages:

1. **Increased Productivity:** Programming languages provide a set of abstractions that allow developers to write code more quickly and efficiently.
2. **Portability:** Programs written in a high-level programming language can run on many different operating systems and platforms.
3. **Readability**: Well-designed programming languages can make code more readable and easier to understand for both the original author and other developers.
4. **Large Community:** Many programming languages have large communities of users and developers, which can provide support, libraries, and tools.

Disadvantages of programming languages:

1. **Complexity**: Some programming languages can be complex and difficult to learn, especially for beginners.
2. **Performance**: Programs written in high-level programming languages can run slower than programs written in lower-level languages.
3. **Limited Functionality**: Some programming languages may not have built-in support for certain types of tasks or may require additional libraries to perform certain functions.
4. **Fragmentation:** There are many different programming languages, which can lead to fragmentation and make it difficult to share code and collaborate with other developers.

Tips for learning new programming language:

1. **Start with the fundamentals**: Begin by learning the basics of the language, such as syntax, data types, variables, and simple statements. This will give you a strong foundation to build upon.
2. **Code daily**: Like any skill, the only way to get good at programming is by practicing regularly. Try to write code every day, even if it's just a few lines.
3. **Work on projects**: One of the best ways to learn a new language is to work on a project that interests you. It could be a simple game, a web application, or anything that allows you to apply what you've learned that is the most important part.
4. **Read the documentation**: Every programming language has documentation that explains its features, syntax, and best practices. Make sure to read it thoroughly to get a better understanding of the language.
5. **Join online communities**: There are many online communities dedicated to programming languages, where you can ask questions, share your code, and get feedback. Joining these communities can help you learn faster and make connections with other developers.
6. **Learn from others**: Find a mentor or someone who is experienced in the language you're trying to learn. Ask them questions, review their code, and try to understand how they solve problems.
7. **Practice debugging**: Debugging is an essential skill for any programmer, and you'll need to do a lot of it when learning a new language. Make sure to practice identifying and fixing errors in your code.

History of Computers

The Evolution of Computers

The history of computers spans thousands of years, from early counting devices to the powerful systems we use today. Here's an overview of the key milestones in the evolution of computers:

1. Early Counting Devices (Pre-Computer Era)

The Abacus (c. 4000 BCE)

The abacus, created by the Chinese, is often regarded as the first computing device. It consisted of beads strung on rods and was used to perform simple arithmetic operations like addition and subtraction. Over time, different versions of the abacus spread across Asia, becoming an essential tool for calculations.

Napier's Bones (1617)

Invented by John Napier, **Napier's Bones** were a set of ivory rods engraved with numbers, designed to assist with multiplication and division. This invention also introduced the concept of the decimal point, a crucial development in simplifying calculations.

2. Mechanical Calculators (17th-19th Century)

Pascaline (1642-1644)

French mathematician **Blaise Pascal** developed the **Pascaline**, the first mechanical calculator capable of performing addition and subtraction. It used gears and wheels to calculate, and its purpose was to help Pascal's father, a tax collector, with his work.

Stepped Reckoner (1673)

German philosopher and mathematician **Gottfried Wilhelm Leibniz** improved Pascal's design, developing the **Stepped Reckoner**. It was capable of performing addition, subtraction, multiplication, and division, and it used fluted drums instead of gears.

Difference Engine (1820s)

Charles Babbage, often called the "Father of Modern Computing," designed the **Difference Engine**, a mechanical device meant to calculate polynomial functions. Though it was never fully built during his lifetime, it demonstrated the potential for automatic computation.

Analytical Engine (1830s)

Babbage also developed the **Analytical Engine**, a more advanced version of the Difference Engine. It was the first design for a general-purpose mechanical computer. It included a control unit, memory, and an input/output system using punch cards. Although it was never constructed, its principles anticipated modern computers.

3. The Rise of Electronic Computing (1930s-1940s)

Tabulating Machine (1890)

Herman Hollerith, an American statistician invented this machine in the year 1890. Tabulating Machine was a mechanical tabulator that was based on punch cards. It was capable of tabulating statistics and record or sort data or information. This machine was used by U.S. Census in the year 1890. Hollerith's Tabulating Machine Company was started by Hollerith and this company later became International Business Machine (IBM) in the year 1924.

Differential Analyzer (1930s)

Differential Analyzer was the first electronic computer introduced in the year 1930 in the United States. It was basically an analog device that was invented by Vannevar Bush. This machine

consists of vacuum tubes to switch electrical signals to perform calculations. It was capable of doing 25 calculations in a few minutes.

Mark I

In the year 1937, major changes began in the history of computers when Howard Aiken planned to develop a machine that could perform large calculations or calculations involving large numbers. In the year 1944, Mark I computer was built as a partnership between IBM and Harvard. It was also the first programmable digital computer marking a new era in the computer world.

4. The Era of Transistors (1950s-1960s)

Transistor Computers (1950s)

In the 1950s, the invention of the **transistor** revolutionized computing. Transistors were smaller, more reliable, and energy-efficient compared to vacuum tubes. They played a key role in making computers more compact and affordable.

UNIVAC I (1951)

The **Universal Automatic Computer I (UNIVAC I)**, developed by **Eckert and Mauchly**, was the first commercially successful computer. It was used for scientific and business applications and demonstrated the potential of electronic computing.

5. The Rise of Integrated Circuits (1960s-1970s)

Integrated Circuits (1960s)

The introduction of **Integrated Circuits (ICs)** allowed multiple transistors to be placed on a single chip, which dramatically reduced the size and cost of computers while improving their performance.

IBM System/360 (1964)

The **IBM System/360** was a family of mainframe computers that utilized integrated circuits, setting a new standard for computing in business, government, and academia. It became one of the first systems to offer compatibility across different machines.

Minicomputers and Microcomputers

With the development of the **microprocessor**, the size of computers shrank even further, leading to the creation of affordable **minicomputers** like the **PDP-8** and **PDP-11**. These smaller systems paved the way for the personal computer revolution.

6. The Personal Computer Revolution (1970s-1980s)

Apple II (1977)

The **Apple II**, developed by **Steve Jobs** and **Steve Wozniak**, was one of the first successful personal computers. It used a microprocessor and could run basic software applications like word processors and games.

IBM PC (1981)

The introduction of the **IBM PC** in 1981 standardized the personal computer market, offering a system that could be easily upgraded and compatible with a wide variety of software. It played a major role in the spread of personal computing.

The Macintosh (1984)

Apple's **Macintosh** introduced the concept of the **graphical user interface (GUI)**, making computers more user-friendly and accessible to a broader audience.

7. The Internet and Networking (1990s-Present)

The World Wide Web (1990s)

The invention of the **World Wide Web** by **Tim Berners-Lee** revolutionized the way people used computers. It made information accessible globally and led to the creation of web browsers like **Netscape Navigator** and **Internet Explorer**.

Cloud Computing (2000s-Present)

Cloud computing allows**have been** users to store and access data remotely via the internet, making it easier to scale computing resources. Services like **Google Drive**, **Dropbox**, and **Amazon Web Services (AWS)** transformed how businesses and individuals manage data.

8. The Modern Day and the Future of Computing

Artificial Intelligence (AI):

AI is rapidly becoming a cornerstone of modern computing. **Machine learning** and **deep learning** algorithms enable computers to make decisions, recognize patterns, and even understand human language, leading to advancements in everything from virtual assistants to autonomous vehicles.

Quantum Computing (Emerging):

Quantum computing promises to revolutionize fields like cryptography and materials science by solving problems that are beyond the reach of classical computers. Though still in its early stages, quantum computers could one day solve complex problems exponentially faster than traditional systems.

The Internet of Things (IoT):

The **Internet of Things (IoT) is** allowed **fifth-generation**, allowing them to collect and share data. From smart homes to wearable tech, IoT devices are transforming the way we interact with the world around us.

Generations of Computers

First Generation Computers

In the period of the year 1940-1956, it was referred to as the period of the first generation of computers. These machines are slow, huge, and expensive. In this generation of computers, vacuum tubes were used as the basic components of CPU and memory. Also, they were mainly dependent on the batch operating systems and punch cards. Magnetic tape and paper tape were used as output and input devices. For example **The**, etc.

Second Generation Computers

The **were** period of the year, 1957-1963 was referred to as the period of the second generation of computers. It was the time of the transistor computers. In the second generation of computers, transistors (which were cheap in cost) are used. Transistors are also compact and consume less power. Transistor computers are faster than first-generation computers. For primary memory, magnetic cores were used, and for secondary memory magnetic disc and tapes for storage purposes. In second-generation computers, COBOL and FORTRAN are used as Assembly language and programming languages, and Batch processing and multiprogramming operating systems are allowed in these computers.

For example **IBM 1620, IBM 7094, CDC 1604, CDC 3600**, etc.

Third Generation Computers

In the third generation of computers, integrated circuits (ICs) were used instead of transistors(in the second generation). A single IC consists of many transistors which increased the power of a computer and also reduced the cost. The third generation computers are more reliable, efficient, and smaller in size. It used remote processing, time-sharing, and multiprogramming as operating systems. FORTRON-II TO IV, COBOL, and PASCAL PL/1 were used which are high-level programming languages.

For example **IBM-360 series, Honeywell-6000 series, IBM-370/168,** etc.

Fourth Generation Computers

The period of 1971-1980 was mainly the time of fourth generation computers. It used VLSI(Very Large Scale Integrated) circuits. VLSI is a chip containing millions of transistors and other circuit elements and because of these chips, the computers of this generation are more compact, powerful, fast, and affordable(low in cost). Real-time, time-sharing and distributed operating system are used by these computers. C and C++ are used as the programming languages in this generation of computers.

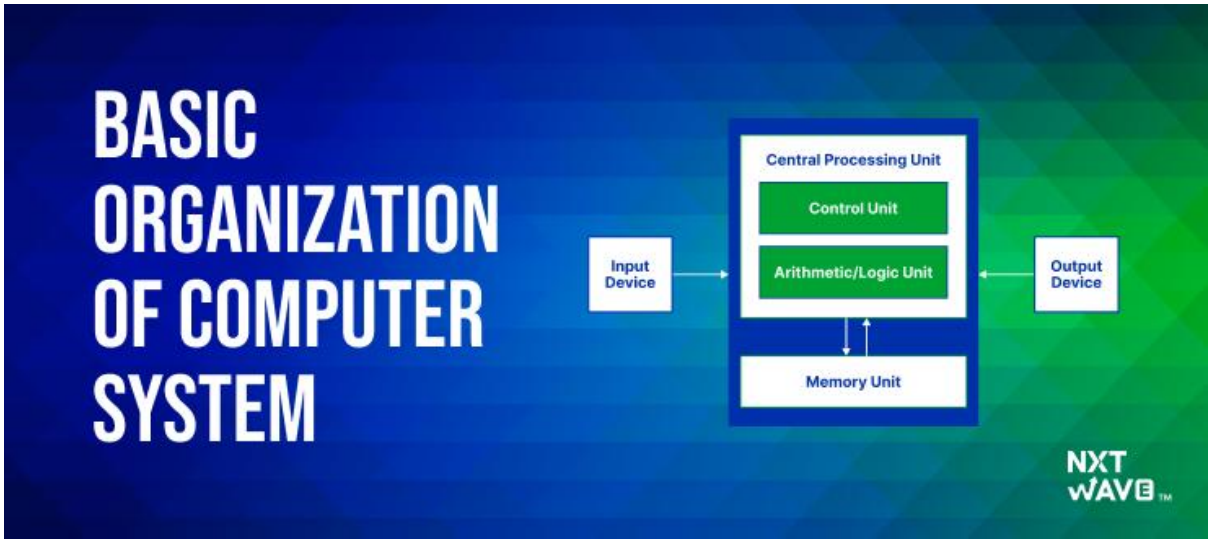
For example **STAR 1000, PDP 11, CRAY-1, CRAY-X-MP,** etc.

Fifth Generation Computers

From 1980 - to till date these computers have been **Internet** used. The ULSI (Ultra Large Scale Integration) technology is used in fifth-generation computers instead of the VLSI technology of fourth-generation computers. Microprocessor chips with ten million electronic components are used in these computers. Parallel processing hardware and AI (Artificial Intelligence) software are also used in fifth-generation computers. The programming languages like C, C++, Java, .Net, etc. are used.

For example **Desktop, Laptop, NoteBook, UltraBook**, etc.

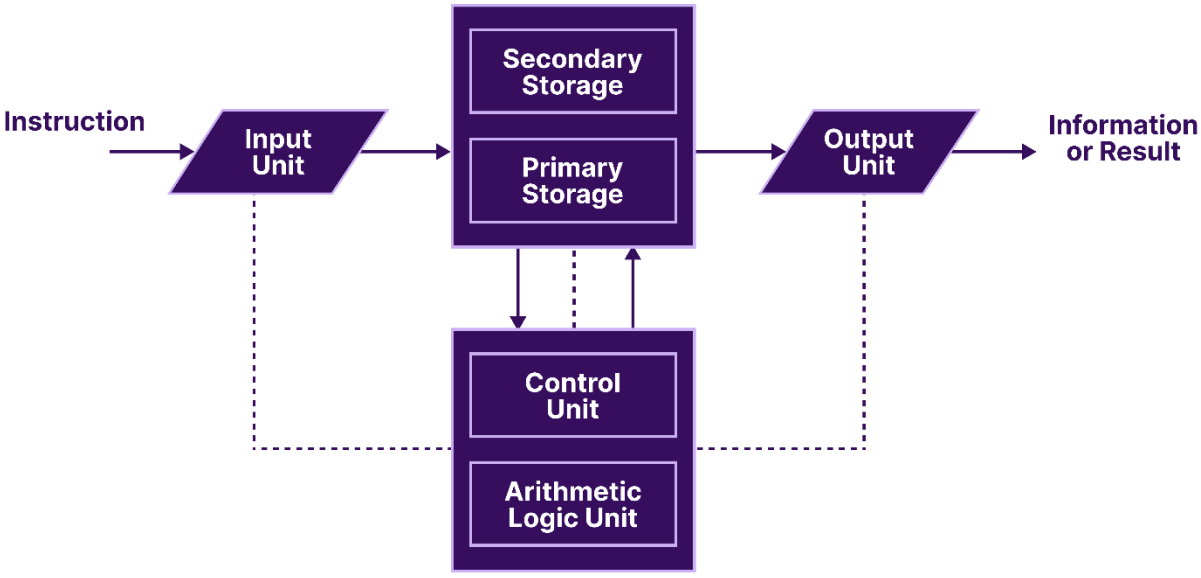
Basic organization of a computer:



At a fundamental level, a computer is a system that takes input, processes it, stores the data temporarily or permanently, and produces output. The organization of a computer is based on Von Neumann Architecture, proposed by John von Neumann in 1945. This architecture divides the computer system into several functional units that work together to execute tasks. These basic components are still used in modern computers, even as their complexity and efficiency have grown significantly.

The key elements in the basic organization of a computer system are:

- Input Unit
- Output Unit
- Control Unit (CU)
- Arithmetic Logic Unit (ALU)
- Memory Unit (Storage)



Central Processing Unit

Functional Components of a Computer

Here are the functional components of a computer:

1. Input Unit

The input unit is responsible for receiving data and instructions from the external environment and converting them into a format that the computer can process. It acts as a bridge between the outside world and the computer system.

Common input devices include:

- Keyboard
- Mouse
- Scanner
- Microphone
- Joystick
- Webcam

2. Output Unit

The output unit provides processed data from the computer system to the external world in a human-readable form. It is the counterpart to the input unit and displays the results of computations.

Common output devices include:

- Monitor
- Printer
- Speakers

3. Control Unit (CU)

The Control Unit (CU) is often described as the 'brain' or 'central nervous system' of the computer. It directs the operations of the entire computer system, including managing the flow of data and instructions.

4. Arithmetic Logic Unit (ALU)

The Arithmetic Logic Unit (ALU) is responsible for performing all the mathematical and logical operations in a computer system. This includes operations like:

- **Arithmetic operations:** Addition, subtraction, multiplication, division.
- **Logical operations:** AND, OR, NOT, XOR.

5. Memory Unit (Storage)

The Memory Unit is used to store data and instructions that are necessary for execution by the CPU. The memory is broadly categorized into two types:

- **Primary Memory (Internal Memory):** This includes RAM and ROM, which are directly accessible by the CPU.
- **Secondary Memory:** This is used for permanent storage and includes devices like hard drives, optical disks, and magnetic tapes.

Difference between Hardware and Software

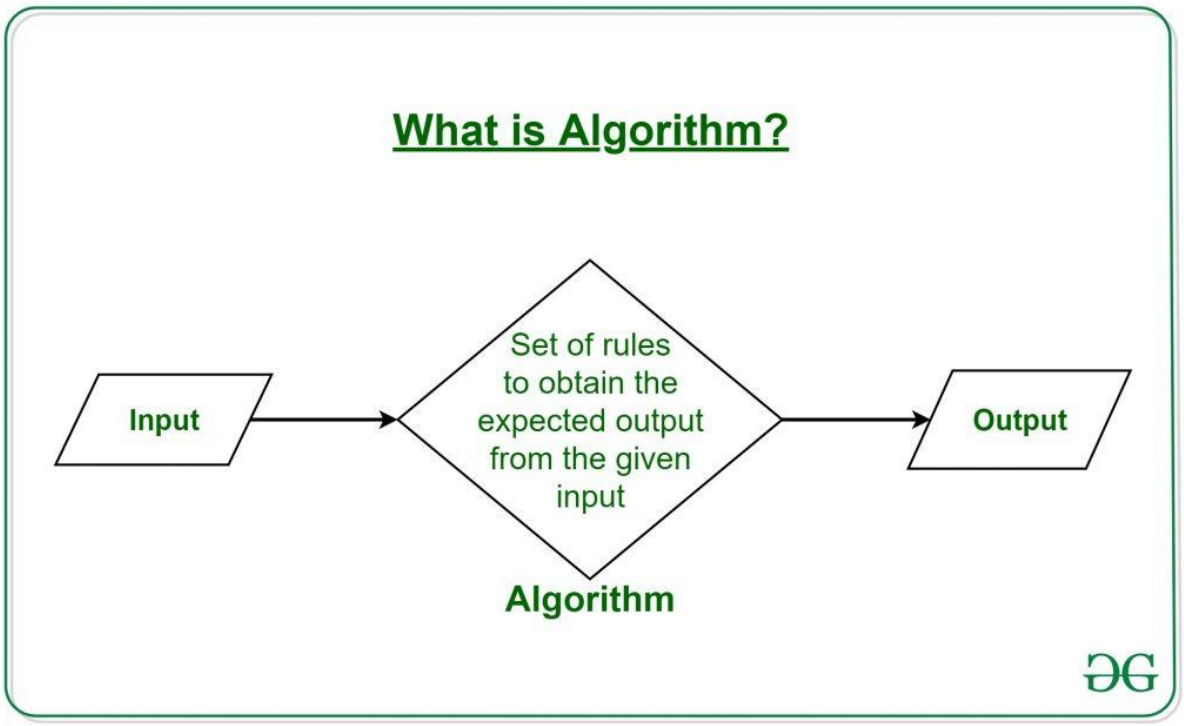
Here is the comparison of Hardware and Software:

Hardware	Software
It defines the physical components that process data.	It is a set of instructions that tell the computer to perform tasks.
It is manufactured using electronic and mechanical materials.	It is developed by writing code in programming languages.
It cannot function without software.	The software needs hardware to run and execute.

Hardware	Software
The hardware is built using metals, plastics, and other materials.	The process of software is developed using programming languages and algorithms.
It is a physical device or sensible gadget (e.g., keyboard, mouse).	It exists in the form of code and files.
The durability of the device lasts for a long time (e.g., hard drive failure, keyboard keys wear out).	It does not physically degrade but may have bugs or glitches over time.
Hardware is not affected by viruses.	Software is vulnerable to viruses, malware, and security flaws.
It cannot be transferred electronically through a network.	It can be easily transferred via a network, cloud, or USB drive.
Hardware operates at machine-level language (binary).	Software is written in high-level programming languages like Java, Python, and C++.
If damaged, hardware must be physically replaced.	Damaged software can be reinstalled or restored from a backup.
The damage caused by physical factors like dust, overheating, humidity, or physical damage.	The damage caused by software bugs, crashes, version conflicts, or overloading.
Examples: Keyboard, mouse, CPU, RAM, hard disk, motherboard, etc.	**Examples:** MS Word, Excel, Photoshop, MySQL, video games, operating systems, etc.

Algorithms Tutorial

Algorithm is a step-by-step procedure for solving a problem or accomplishing a task. In the context of data structures and algorithms, it is a set of well-defined instructions for performing a specific computational task. Algorithms are fundamental to computer science and play a very important role in designing efficient solutions for various problems. Understanding algorithms is essential for anyone interested in mastering data structures and algorithms.



What is an Algorithm?

An **algorithm** is a finite sequence of well-defined instructions that can be used to solve a computational problem. It provides a step-by-step procedure that convert an input into a desired output.

Algorithms typically follow a logical structure:

- **Input:** The algorithm receives input data.
- **Processing:** The algorithm performs a series of operations on the input data.
- **Output:** The algorithm produces the desired output.

What is the Need for Algorithms?

Algorithms are essential for solving complex computational problems efficiently and effectively. They provide a systematic approach to:

- **Solving problems:** Algorithms break down problems into smaller, manageable steps.
- **Optimizing solutions:** Algorithms find the best or near-optimal solutions to problems.
- **Automating tasks:** Algorithms can automate repetitive or complex tasks, saving time and effort.

Introduction to Flowcharts

Flowcharts are the visual representations of an algorithm or a process. Flowcharts use symbols/shapes like arrows, rectangles, and diamonds to properly explain the sequence of steps involved in the algorithm or process. Flowcharts have their use cases in various fields such as software development, business process modeling, and engineering.

Why use Flowcharts?

Flowcharts are used due to the numerous amount of benefits they provide. Below are some of the important reasons to use flowcharts:

- They provide clarity and simplification to the complex processes and algorithms, which in turn helps other people to understand them easily.
- Flowcharts provide a universal visual language that can be understood by anyone across different teams and helps reduce miscommunications.
- They are an optimal solution for documenting standard operating procedures, workflows, or business processes. This makes it easier to train new employees.
- Flowcharts help in increasing the visualization of the problem being solved which enables more informed and data-driven choices.

Types of Flowcharts

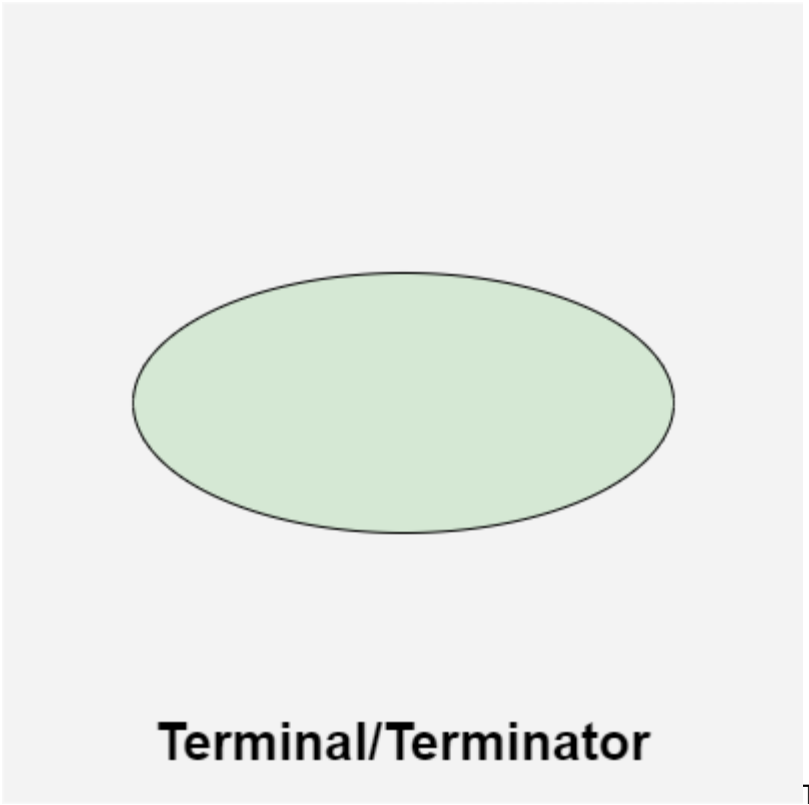
There are many types of flowcharts, each is designed to represent different kinds of processes and information. Some common types of flowcharts are:

- **Process Flowchart:** It represents the sequence of steps in a process. They are frequently used in business process modeling, manufacturing, and project management
- **Swimlane Flowchart:** It organizes the process into different lanes, each representing a different person or department and is used for illustrating how different teams or departments collaborate within a process
- **Workflow Diagram:** It represents how tasks, documents, or information move through a system and is commonly used in office processes or software development
- **Data Flow Diagram (DFD):** It focuses on detailing the inputs, processes, and outputs. Used in system design and analysis to model the flow of data within a system
- **Decision Flowchart:** It focuses on mapping out decision points within a process and the possible outcomes of each decision. It is used in decision-making scenarios

Symbols used in Flowchart Designs

1. Terminal/Terminator

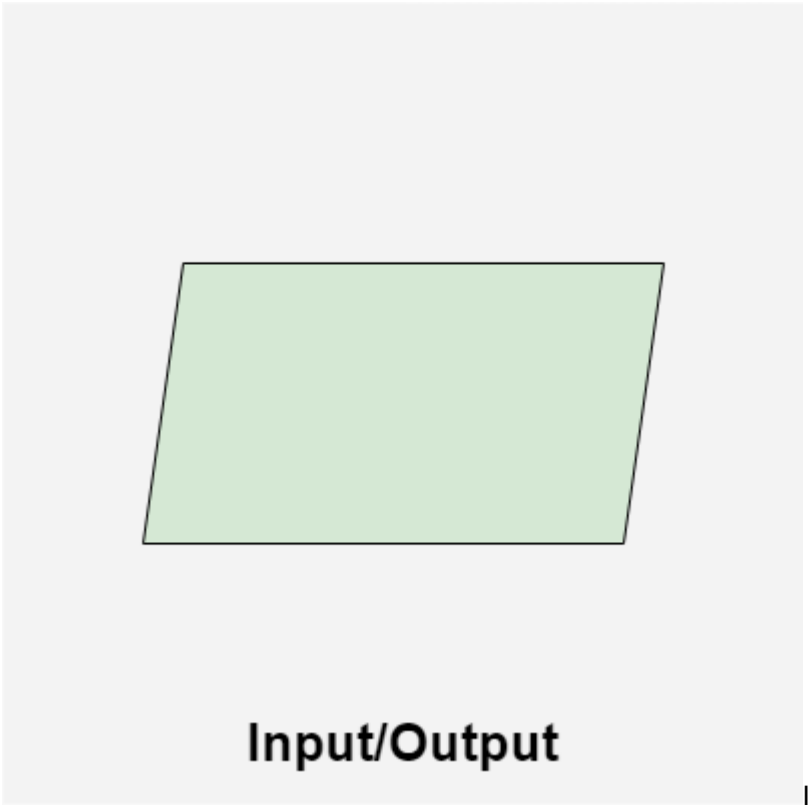
The oval symbol indicates Start, Stop and Halt in a program's logic flow. A pause/halt is generally used in a program logic under some error conditions. Terminal is the first and last symbols in the flowchart.



Terminal/Terminator

2. Input/Output

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



Input/Output

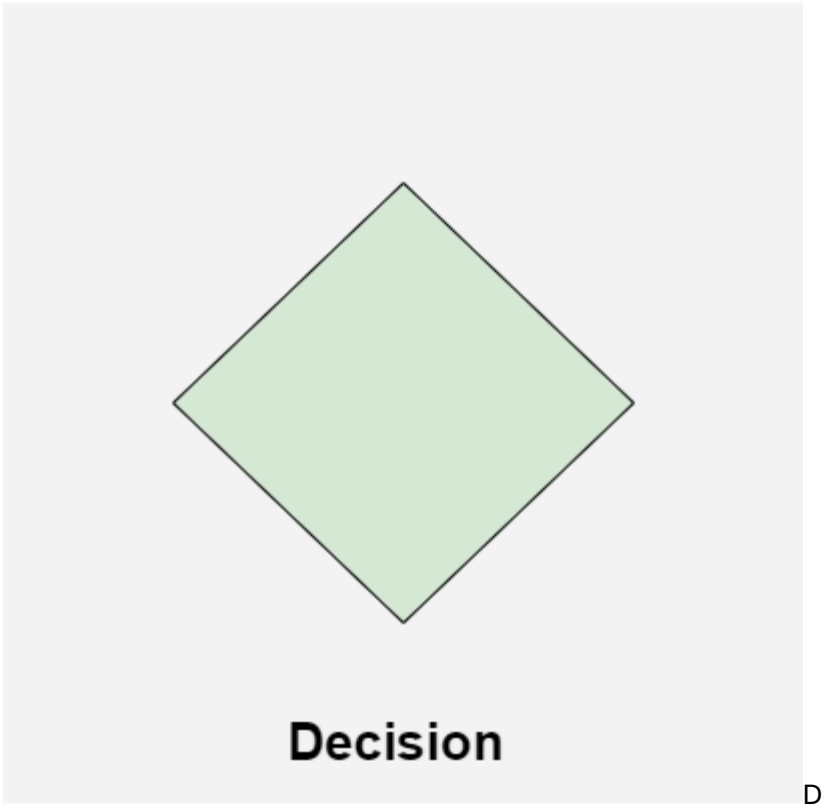
3. Action/Process

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

Action/Process

4. Decision

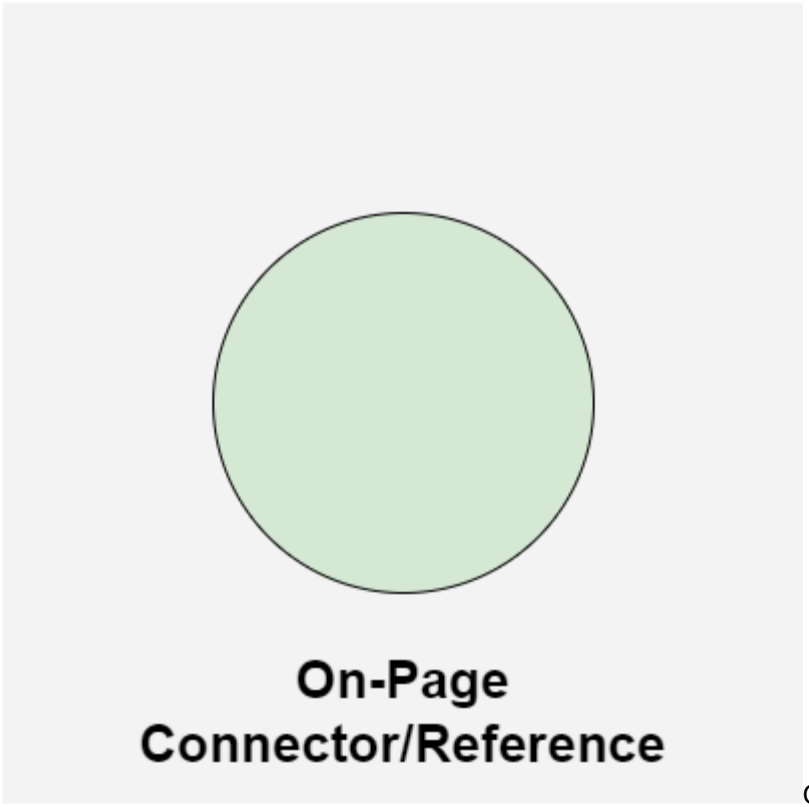
Diamond symbol represents a decision point. Decision based operations such as yes/no question or true/false are indicated by diamond in flowchart.



Decision

5. On-Page Connector/Reference

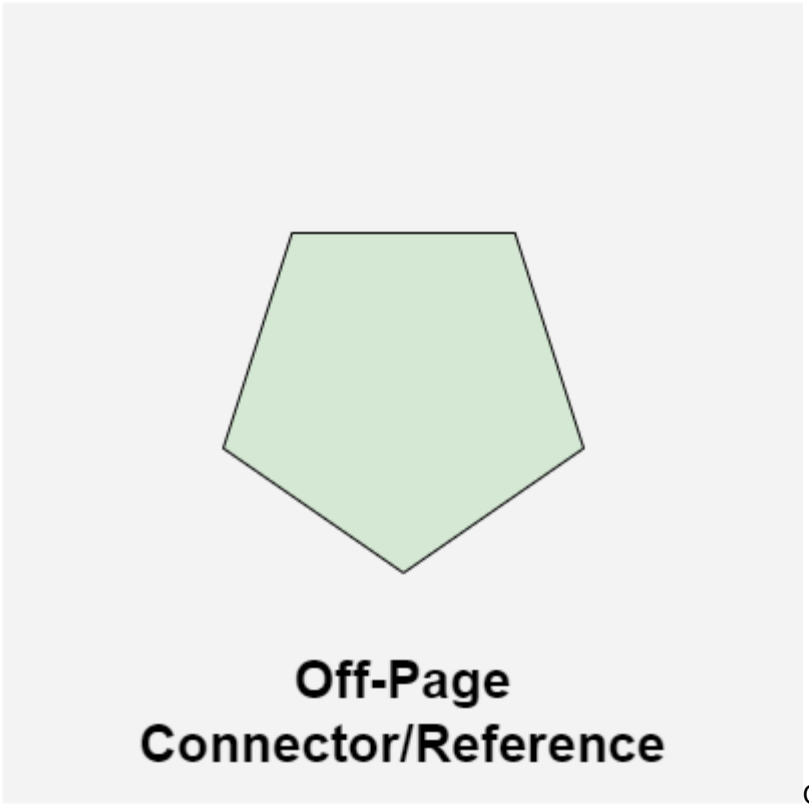
Whenever flowchart becomes complex or it spreads over more than one page, it is useful to use connectors to avoid any confusions. connectors are used to indicate a jump from one part of the flowchart to another without drawing long or complicated lines. On-Page Connector is represented by a small circle



On-Page Connector/Reference

6. Off-Page Connector/Reference

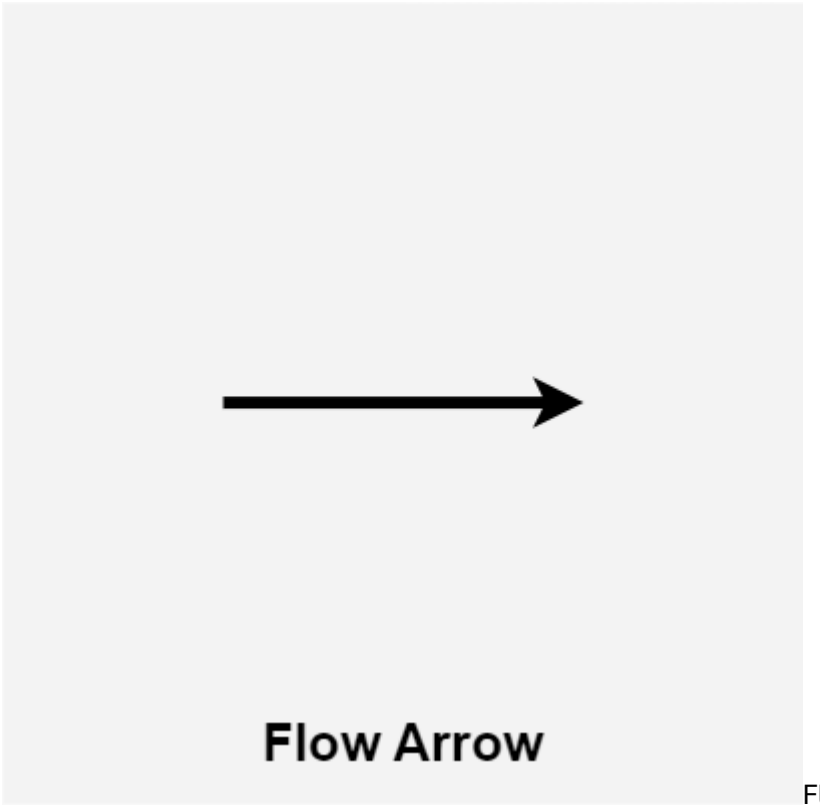
Whenever flowchart becomes complex or it spreads over more than one page, it is useful to use connectors to avoid any confusions. connectors are used to indicate a jump from one part of the flowchart to another without drawing long or complicated lines. Off-Page Connector is represented by a pentagon.



Off-Page Connector/Reference

7. Flow lines

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



Flow lines

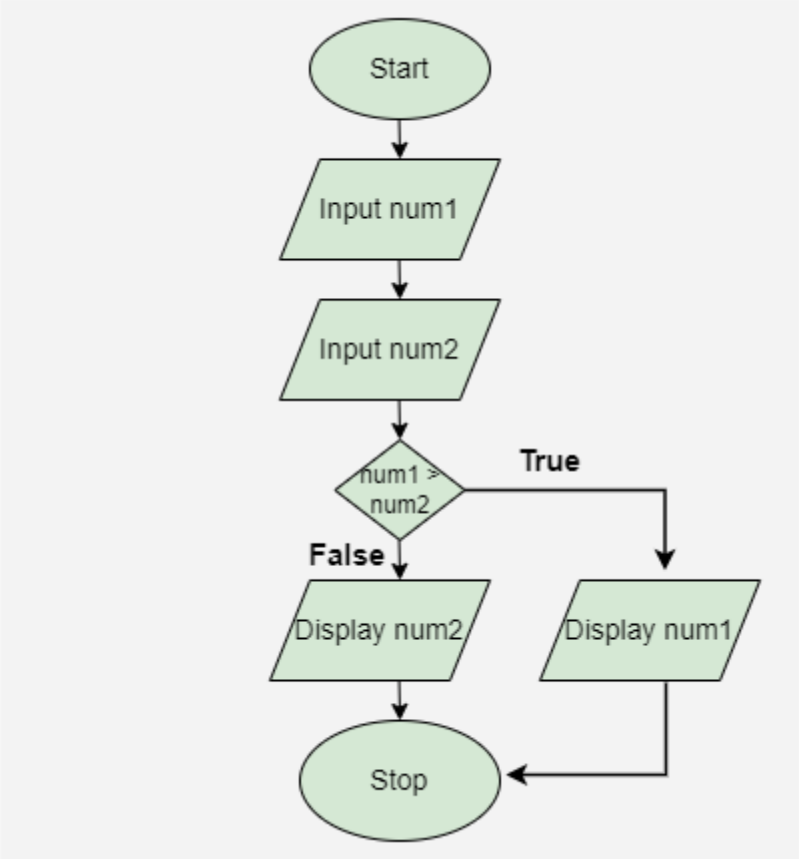
Rules For Creating a Flowchart

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

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

Example of a Flowchart

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



Example Flowchart

Below is the explanation of the above flowchart:

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

    - If **False**, the process moves to display **num2**.
- **Stop**: The process ends with the **Stop** symbol, signaling the conclusion of the program.

Advantages of using a Flowchart

- Flowcharts are a better way of communicating the logic of the system.
- Flowcharts act as a guide for blueprint during program designed.
- Flowcharts help in debugging process.
- With the help of flowcharts programs can be easily analyzed.
- They provide better documentation.
- Flowcharts serve as a good proper documentation.

Disadvantages of using a Flowchart

- It is difficult to draw flowcharts for large and complex programs.
- There is no standard to determine the amount of detail.
- It is very difficult to modify the Flowchart.
- Making a flowchart is costly.
- If changes are done in software, then the flowchart must be redrawn

how to install Dia and then make use of this outstanding, open source tool.

Installation

The installation of Dia is simple. Like all good modern Linux distributions, I am sure the one you are using has a robust Add/Remove Software tool like Synaptic. These tools make installing a snap. For example, to install Dia with Synaptic follow these steps:

1. Open up the Synaptic tool.
2. Search for “dia” (no quotes).
3. Mark Dia for installation.
4. Click Apply to install.

That’s it. Of course, depending upon your installation, some dependencies will have to be resolved (which Synaptic should handle smoothly). If your Linux machine doesn’t have Synaptic, it will have a similar front-end for PackageKit (look for Add/Remove Software in your menu).

You can also opt to install Dia from command line. This too is simple. If you are using an apt-based system (such as Ubuntu), you can install with the command:

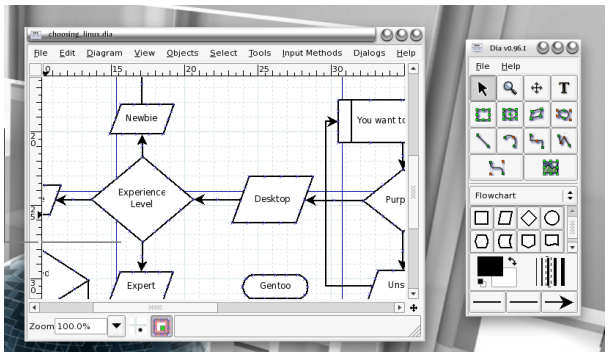
sudo apt-get install dia

If you are using an rpm-based system (such as Fedora), you can install with the command:

yum install dia

Now that you are installed, you are ready to start diagramming.

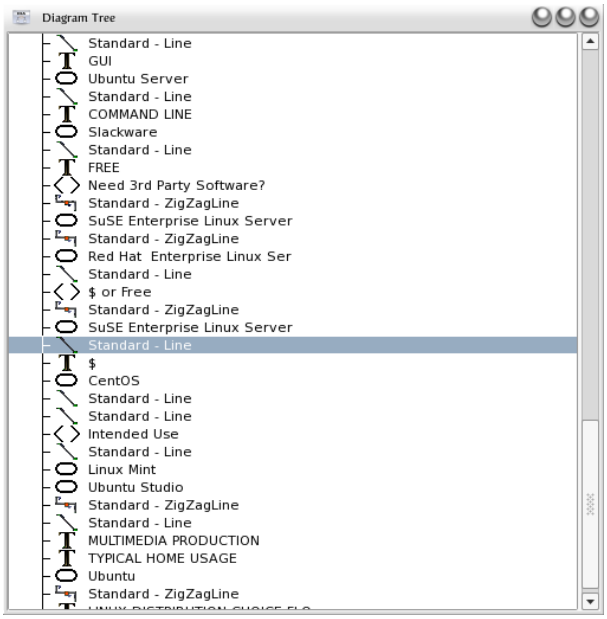
Interface



Dia has a somewhat similar interface to The GIMP. By that, I mean Dia uses one window for the working window, and a separate window for the Toolbox (see Figure 1). The left window is the working window (where you create your charts) and the right window is the toolbox.

It is from the toolbox that you select the tools and elements you want to use as well as opening/saving/exporting files and so on.

There is a third window that many do not know about. This is the Diagram Tree window. When you have your flowchart open click **File > Diagram Tree** the Diagram Tree window will open (see Figure 2). This window lists all of the elements in your flow chart. You can double-click an element and your working window will automatically zoom in to the location of that element. This is especially helpful when dealing with larger flowcharts (in both size and scope).



But how to begin? The fundamentals of flow charts are beyond the scope of this article. If you are unsure how to create a flowchart, I suggest you fire up everyone's favorite online classroom (Google that is) and learn a thing or two about how flowcharts are created. Once you are armed with that knowledge, you are ready to start working.

The first thing to do (once you have Dia opened up) is to select the type of elements you are going to need to work with. From the drop-down list (in the center of the Toolbox window), you can select from:

- Flowchart: Typical elements for standard flowcharts.
- Assorted: These are elements that can't be placed in specific categories. This list includes various geometric shapes as well as arrows.
- UML: Unified Modeling Language. This is for the workflow of step-wise activities (such as the step-by-step work flow of components in a system).
- MISC: This contains only four elements: Folder, file, Clock, and group-testing object.

There is also an "Other Sheets" menu that contains numerous entries such as:

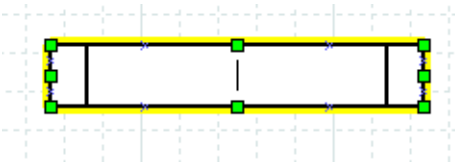
- Cybernetics
- ER

- Jigsaw
- Network
- Pneumatic
- Sybase

And many more.

After you have selected the type of diagram you are to create you can then select the element that will begin your chart. Instead of walking you through the entire process of creating a chart, I am only going to show you the steps that might cause you confusion.

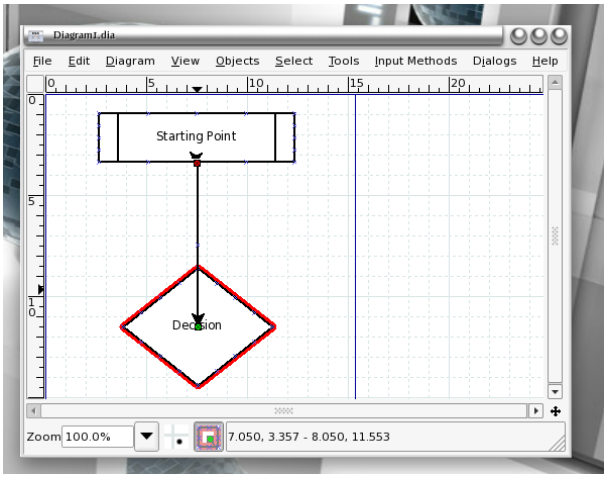
Placing an Element



When you select the element (from the toolbox) you want to use go to the Working window and place the cursor where you want to insert the element. Click and drag your cursor until that element is the size you desire. Once you have placed that element, a new text cursor will appear (see Figure 3) where you can now type the text for this element. Once you have typed the necessary text just click outside of that element to set the text. If you hit the Enter key you will add a carriage return to your text which will, in turn, make your element taller.

Adding a Line

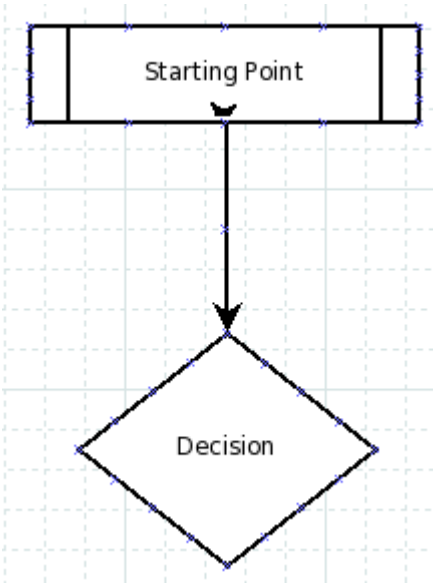
What would a flowchart be without connecting lines to help show the flow of your chart? In order to add a line in Dia you first need two elements to connect. To make things simple, let's add a standard line going from a Starting Point to a Decision. To do this follow these steps:



Select the Line tool from the Toolbox window.

Left-click on the Starting point element and hold the mouse button down.

Drag the cursor to the next element and release (see Figure 4).



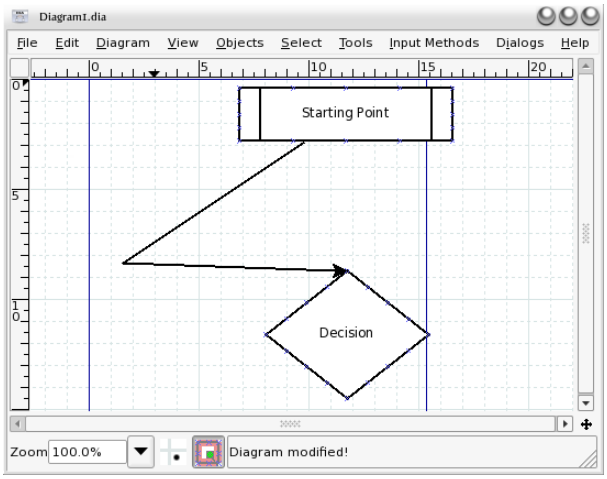
The line will automatically connect the two elements. You do not have to worry about placing the cursor at the edges of the elements as Dia will detect them for you. Figure 5 shows the two elements with the line properly connecting them. As you can see, the line is set in place perfectly.

The line will also automatically place directional arrows. If you wanted the direction of that line to go the opposite direction you would just have to start the line from the other element. Simple.

One very nifty trick is that once two objects are connected with a line, you can move those objects around and they will remain connected. The line will stretch and angle to match wherever you place the elements.

Polyline

The last thing to mention would be polylines. Polylines allow you to add corners to lines so you can be as precise as needed in the placement of your lines and elements. To create a Polyline, click the Polyline tool in the toolbox and then draw a line as you would normally. Now right click that line and select Add Corner. The new corner will show up as a red dot in the line. You can now click and drag that corner to any place you need so to create the exact line shape (see Figure 6).



Final Thoughts

You will be surprised how simple Dia is to use to create just about any type of flow chart you need. Although this tutorial only scratches the surface of what you can do with Dia, it will get you started on what will become a very rewarding relationship between you and a very helpful open source application.

What is PseudoCode

*A **Pseudocode** is defined as a step-by-step description of an algorithm. Pseudocode does not use any programming language in its representation instead it uses the simple English language text as it is intended for human understanding rather than machine reading.*
*Pseudocode is the **intermediate state between an idea and its implementation(code)** in a high-level language.*

What is the need for Pseudocode

Pseudocode is an important part of designing an algorithm, it helps the programmer in planning the solution to the problem as well as the reader in understanding the approach to the problem.

Pseudocode is an intermediate state between algorithm and program that plays supports the transition of the algorithm into the program.

Pseudocode is an intermediate state between algorithm and program

How to write Pseudocode?

Before writing the pseudocode of any algorithm the following points must be kept in mind.

- Organize the sequence of tasks and write the pseudocode accordingly.
- At first, establishes the main goal or the aim. **Example:**

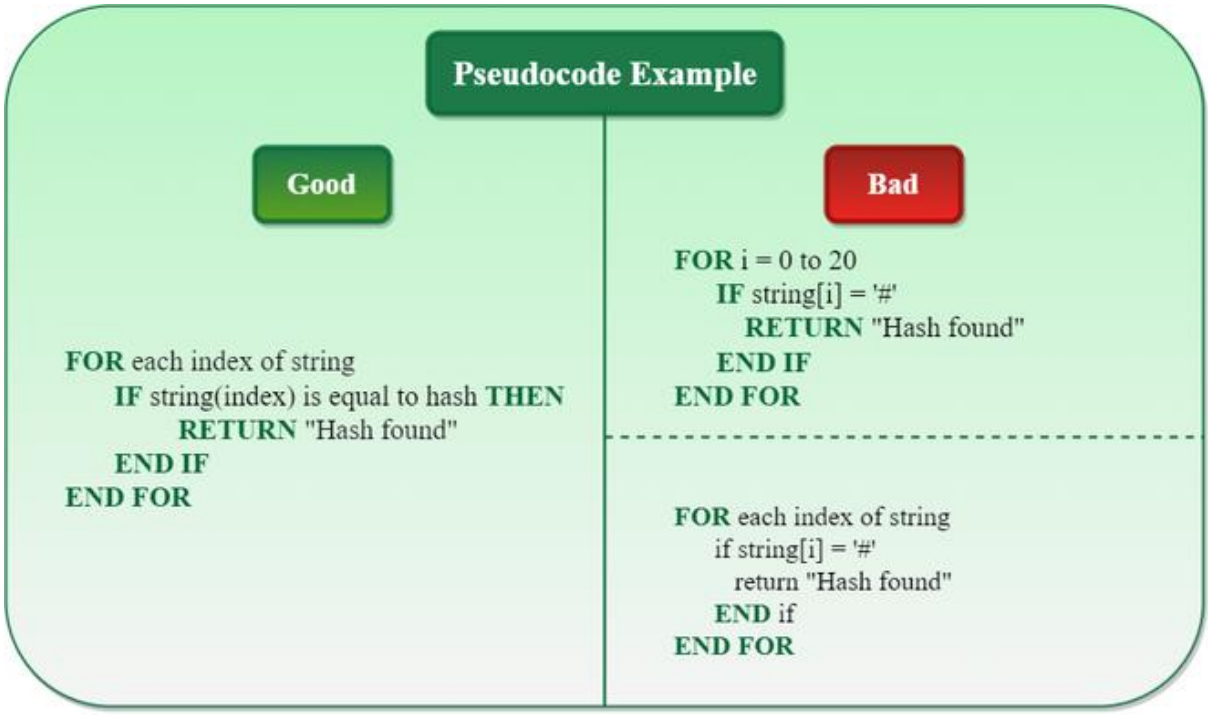
*This program will print first **N** numbers of Fibonacci series.*

- Use standard programming structures such as **if-else**, **for**, **while**, and **cases** the way we use them in programming. Indent the statements if-else, for, while loops as they are indented in a program, it helps to comprehend the decision control and execution mechanism. It also improves readability to a great extent. **Example:**

```
IF "1"
  print response
    "I AM CASE 1"

IF "2"
  print response
    "I AM CASE 2"
```

- Use appropriate naming conventions. The human tendency follows the approach of following what we see. If a programmer goes through a pseudo code, his approach will be the same as per that, so the naming must be simple and distinct.
- Reserved commands or keywords must be represented in **capital letters**. **Example:** if you are writing IF...ELSE statements then make sure IF and ELSE be in capital letters.
- Check whether all the sections of a pseudo code are complete, finite, and clear to understand and comprehend. Also, explain everything that is going to happen in the actual code.
- Don't write the pseudocode in a programming language. It is necessary that the pseudocode is simple and easy to understand even for a layman or client, minimizing the use of technical terms.

Good vs Bad ways of writing Pseudocode:



Compilation and Execution

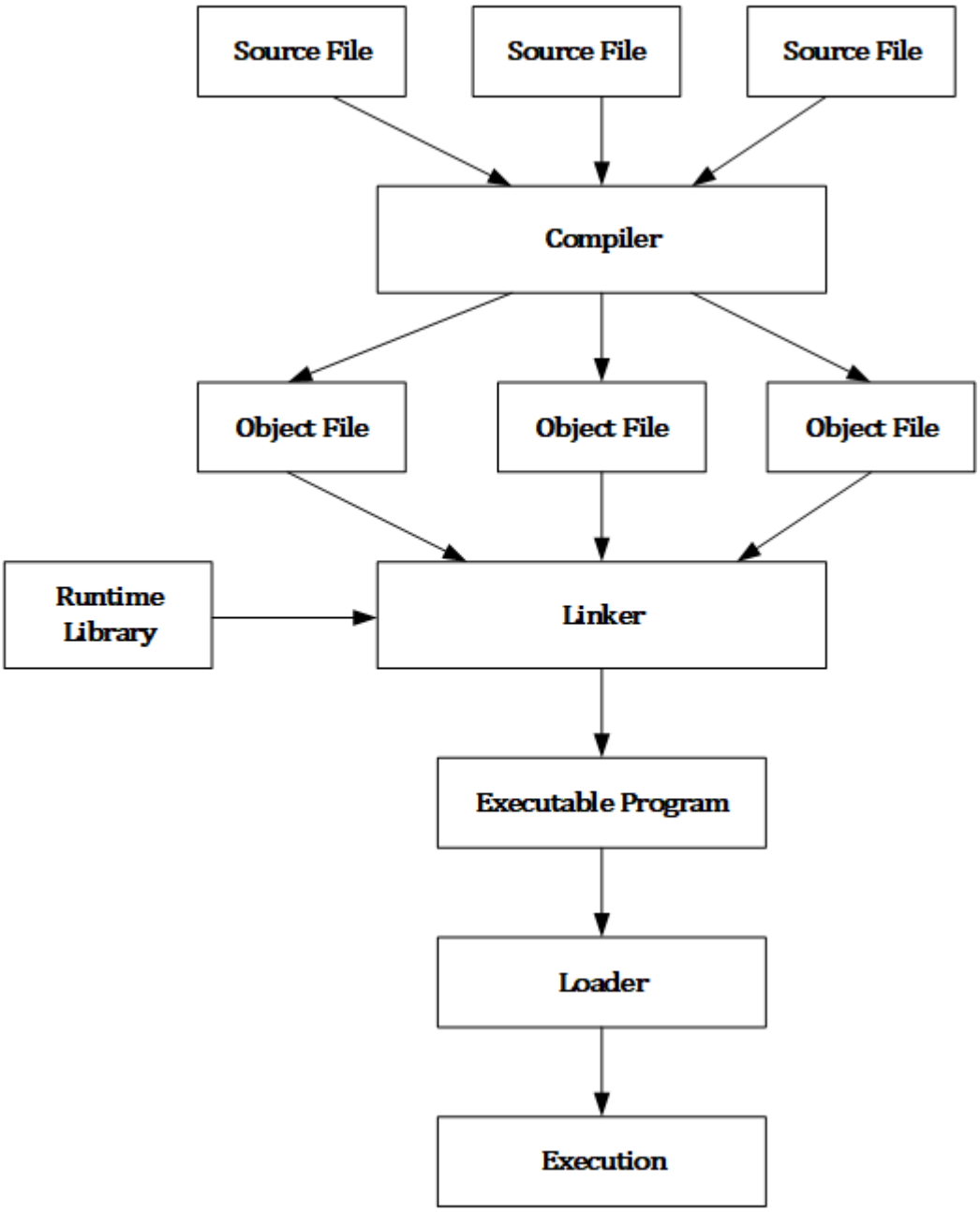
Compilation and Execution Process

Generally coding is done in high level language or low level language (assembly language). For the computer to understand these languages, they must be translated into machine level language. The translation process is carried out by a compiler/interpreter (for high level language) or an assembler (for assembly language program). The machine language code thus created can be saved and run immediately or later on.

In an interpreted program, each program statement is converted into machine code before program is executed. The execution occurs immediately one statement at a time sequentially. BASIC is one of the frequently used interpreted language. In contrast to interpreter, a compiler converts a given source code into object code. Once an object code is obtained, the compiled programs can be faster and more efficient than interpreted programs.

Compilation Process

A source code must go through several steps before it becomes an executable program. In the first step the source code is checked for any syntax errors. After the syntax errors are traced out a source file is passed through a compiler which first translates high level language into object code (A machine code not ready to be executed). A linker then links the object code with pre-compiled library functions, thus creating an executable program. This executable program is then loaded into the memory for execution. General compilation process is shown in Figure below:



C Data Types

Data types in C programming language enables the programmers to appropriately select the data as per requirements of the program and the associated operations of handling it.

Data types in c language can be broadly classified as:

1. **Primitive Data Types**
2. **User Defined Data Types**, for example, enum, structure, union
3. **Derived Data Types**, for example, array, pointers

In this tutorial we will only focus on primitive data types, user defined and derived data types will be discussed separately.

Primitive Data Types

The primitive data types in c language are the inbuilt data types provided by the c language itself. Thus, all c compilers provide support for these data types.

The following primitive data types in c are available:

Integer Data Type, int

Integer data type is used to declare a variable that can store numbers without a decimal. The keyword used to declare a variable of integer type is "int". Thus, to declare integer data type following syntax should be followed:

int variable_name;

Float data Type, float

Float data type declares a variable that can store numbers containing a decimal number.

Syntax

float variable_name;

Double Data Type, double

Double data type also declares variable that can store floating point numbers but gives precision double than that provided by float data type. Thus, double data type are also referred to as double precision data type.

Syntax

double variable_name;

Character Data Type, char

Character data type declares a variable that can store a character constant. Thus, the variables declared as char data type can only store one single character.

Syntax

char variable_name;

Void Data Type, void

Unlike other primitive data types in c, void data type does not create any variable but returns an empty set of values. Thus, we can say that it stores null.

Syntax

void variable_name;

Data Type Qualifiers

Apart from the primitive data types mentioned above, there are certain data type qualifiers that can be applied to them in order to alter their range and storage space and thus, fit in various situations as per the requirement.

The data type qualifiers available in c are:

- short
- long
- signed
- unsigned

It should be noted that the above qualifiers cannot be applied to float and can only be applied to integer and character data types.

The entire list of data types in c available for use is given below:

C Data Types		**Size(in bytes)**	**Range**
Integer Data Types	int signed int unsigned int short int signed short int unsigned short int long int signed long int unsigned long int	2 2 2 1 1 1 4 4 4	-32,768 to 32,767 -32,768 to 32,767 0 to 65535 -128 to 127 -128 to 127 0 to 255 -2,147,483,648 to 2,147,483,647 Same as Above 0 to 4,294,967,295
Floating Point Data Types	float double long double	4 8 10	3.4E-38 to 3.4E+38 1.7E-308 to 1.7E+308 3.4E-4932 to 1.1E+4932
Character Data Types	char signed char unsigned char	1 1 1	-128 to 127 -128 to 127 0 to 255

C Variables, Constants and Literals

Variables

In programming, a variable is a container (storage area) to hold data.

To indicate the storage area, each variable should be given a unique name (identifier). Variable names are just the symbolic representation of a memory location. For example:

int age = 25;

Here, age is a variable of int type and we have assigned an integer value 25 to it.

The value of a variable can be changed, hence the name variable.

char ch = 'a';

// some code

ch = 'l';

Constants

If you want to define a variable whose value cannot be changed, you can use the const keyword. This will create a constant. For example,

const double PI = 3.14;

Notice, we have added keyword const.

Here, PI is a symbolic constant; its value cannot be changed.

const double PI = 3.14;

PI = 2.9; //Error

Literals

Literals are data used for representing fixed values. They can be used directly in the code. For example: 1, 2.5, 'c' etc.

Here, 1, 2.5 and 'c' are literals. Why? You cannot assign different values to these terms.

---

1. Integers

An integer is a numeric literal(associated with numbers) without any fractional or exponential part. There are three types of integer literals in C programming:

- decimal (base 10)
- octal (base 8)
- hexadecimal (base 16)

For example:

Decimal: 0, -9, 22 etc

Octal: 021, 077, 033 etc

Hexadecimal: 0x7f, 0x2a, 0x521 etc

In C programming, octal starts with a 0, and hexadecimal starts with a 0x.

---

2. Floating-point Literals

A floating-point literal is a numeric literal that has either a fractional form or an exponent form. For example:

-2.0

0.0000234

-0.22E-5

Note: E-5 = 10^{-5}

---

3. Characters

A character literal is created by enclosing a single character inside single quotation marks. For example: 'a', 'm', 'F', '2', '}' etc.

---

4. String Literals

A string literal is a sequence of characters enclosed in double-quote marks. For example:

"good" //string constant

"" //null string constant

" " //string constant of six white space

"x" //string constant having a single character.

"Earth is round\n" //prints string with a newline

---

5. Escape Sequences

Sometimes, it is necessary to use characters that cannot be typed or has special meaning in C programming. For example: newline(enter), tab, question mark etc.

In order to use these characters, escape sequences are used.

Escape Sequences	Character

\b	Backspace
\f	Form feed
\n	Newline
\r	Return
\t	Horizontal tab
\v	Vertical tab
\\	Backslash
\'	Single quotation mark
\"	Double quotation mark
\?	Question mark
\0	Null character

Basic Input and Output in C

In C programming, input and output operations refer to reading data from external sources and writing data to external destinations outside the program. C provides a standard set of functions to handle input from the user and output to the screen or to files. These functions are part of the standard input/output library **<stdio.h>**.

In C, there are many functions used for input and output in different situations, but the most commonly used functions for Input/Output are scanf() and printf(), respectively.

Basic Output in C

In C, there are many functions used for output in different situations, but the most commonly used function for output is printf().

The **printf()** function is used to print formatted output to the standard output stdout (which is generally the console screen). It is one of the most commonly used functions in C.

Syntax

printf("formatted_string", variables/values);

Where,

- **Formatted String:** string defining the structure of the output and include **format specifiers**
- **variables/values**: arguments passed to printf() that will replace the format specifiers in the formatted string.

Examples

The following examples demonstrate the use of printf for output in different cases:

Printing Some Text

```
#include <stdio.h>


int main() {


  // Prints some text

  printf("First Print");


  return 0;

}
```

Output

First Print

Explanation: The text inside "" is called a string in C. It is used to represent textual information. We can directly pass strings to the printf() function to print them in console screen.

Printing Variables

```
#include <stdio.h>


int main() {

  int age = 22;


  // Prints Age

  printf("%d\n", age);
```

```
    return 0;
}
```

Output

22

Here, the value of variable age is printed. You may have noticed **%d** in the formatted string. It is actually called format specifier which are used as placeholders for the value in the formatted string.

You may have also noticed '\n' character. This character is an escape sequence and is used to enter a newline.

Printing Variables Along with String

```
#include <stdio.h>


int main() {
    int age = 22;


    // Prints Age
    printf("The value of the variable age is %d\n", age);


    return 0;
}
```

Output

The value of the variable age is 22

The printf function in C allows us to format the output string to console. This type of string is called formatted string as it allows us to format (change the layout the article).

fputs()

The fputs() function is used to output strings to the files but we can also use it to print strings to the console screen.

Syntax:

```
fputs("your text here", stdout);
```

Where, the **stdout** represents that the text should be printed to console.

Example

```
#include <stdio.h>


int main() {
   fputs("This is my string", stdout);
   return 0;
}
```

Output

This is my string

Basic Input in C

These functions provide ways to read data from the user and use it in the programs . Among the most commonly used input functions are scanf() for reading formatted data and getchar() for reading a single character.

scanf() is used to read user input from the console. It takes the format string and the addresses of the variables where the input will be stored.

Syntax

```
scanf("formatted_string", address_of_variables/values);
```

Remember that this function takes the address of the arguments where the read value is to be stored.

Examples of Reading User Input

The following examples demonstrate how to use the scanf for different user input in C:

Reading an Integer

```
#include <stdio.h>


int main() {
   int age;
   printf("Enter your age: ");

   // Reads an integer
   scanf("%d", &age);

   // Prints the age
```

```
    printf("Age is: %d\n", age);
    return 0;
}
```

Output

Enter your age:
25 *(Entered by the user)*
Age is: 25

Explanation: %d is used to read an integer; and &age provides the address of the variable where the input will be stored.

Reading a Character

```
#include <stdio.h>

int main() {
    int ch;
    printf("Enter a character: \n");

    // Reads an integer
    scanf("%c", &ch);

    // Prints the age
    printf("Entered character is: %d\n", ch);
    return 0;
}
```

Output

Enter a character:
a (Entered by the user)
Entered character is: a

Reading a string

The scanf() function can also be used to read string input from users. But it can only read single words.

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

```
int main() {
    char str[100];  // Declare an array to hold the input string

    printf("Enter a string: ");
    scanf("%s", str);  // Reads input until the first space or newline

    printf("You entered: %s\n", str);

    return 0;
}
```

Output:

Enter a String:
Geeks (Entered by the user)
Entered string is: Geeks

The scanf() function can not handle spaces and stops at the first blanksspace. to handle this situation we can use fgets() which is a better alternative as it can handle spaces and prevent buffer overflow.

fgets()

fgets() reads the given number of characters of a line from the input and stores it into the specified string. It can read multiple words at a time.

Syntax

fgets(str, n, stdin);

where **buff** is the string where the input will be stored and **n** is the maximum number of characters to read. **stdin** represents input reading from the keyboard.

Example:

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

int main() {

    // String variable
    char name[20];
```

```
    printf("Enter your name: \n");
    fgets(name, sizeof(name), stdin);


    printf("Hello, %s", name);
    return 0;
}
```

Output

Enter your name:
Shubham Kumar *(Entered by User)*
Hello, Shubham Kumar

Type Casting and Conversion in C

In a programming language, the expression contains data values of the same datatype or different data types. When the expression contains similar datatype values then it is evaluated without any problem. But if the expression contains two or more different datatype values then they must be converted to the single datatype of destination datatype. Here, the destination is the location where the final result of that expression is stored. For example, the multiplication of an integer data value with the float data value and storing the result into a float variable. In this case, the integer value must be converted to float value so that the final result is a float datatype value.
In a c programming language, the data conversion is performed in two different methods as follows...

1. Type Conversion
2. Type Casting

Type Conversion

The type conversion is the process of converting a data value from one data type to another data type automatically by the compiler. Sometimes type conversion is also called **implicit type conversion**. The implicit type conversion is automatically performed by the compiler.
For example, in c programming language, when we assign an integer value to a float variable the integer value automatically gets converted to float value by adding decimal value 0. And when a float value is assigned to an integer variable the float value automatically gets converted to an integer value by removing the decimal value. To understand more about type conversion observe the following...

```
int i = 10 ;
float x = 15.5 ;
char ch = 'A' ;

i = x ; =======> x value 15.5 is converted as 15 and assigned to variable i
```

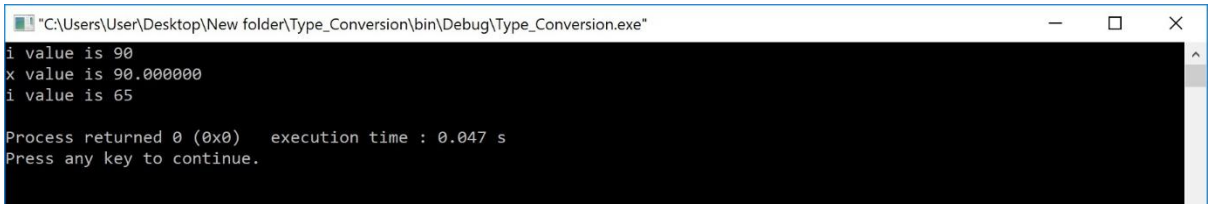
x = i ; =======> Here i value 10 is converted as 10.000000 and assigned to variable x

i = ch ; =======> Here the ASCII value of A (65) is assigned to i

Example Program

```
#include<stdio.h>

#include<conio.h>


void main(){

  int i = 95 ;

  float x = 90.99 ;

  char ch = 'A' ;


  i = x ;

  printf("i value is %d\n",i);

  x = i ;

  printf("x value is %f\n",x);

  i = ch ;

  printf("i value is %d\n",i);


}
```

Output:



In the above program, we assign **i = x**, i.e., float variable value is assigned to the integer variable. Here, the compiler automatically converts the float value (90.99) into integer value (90) by removing the decimal part of the float value (90.99) and then it is assigned to variable **i**. Similarly, when we assign **x = i**, the integer value (90) gets converted to float value (90.000000) by adding zero as the decimal part.

Typecasting

Typecasting is also called an **explicit type conversion**. Compiler converts data from one data type to another data type implicitly. When compiler converts implicitly, there may be a data

loss. In such a case, we convert the data from one data type to another data type using explicit type conversion. To perform this we use the **unary cast operator**. To convert data from one type to another type we specify the target data type in parenthesis as a prefix to the data value that has to be converted. The general syntax of typecasting is as follows.

(TargetDatatype) DataValue

Example

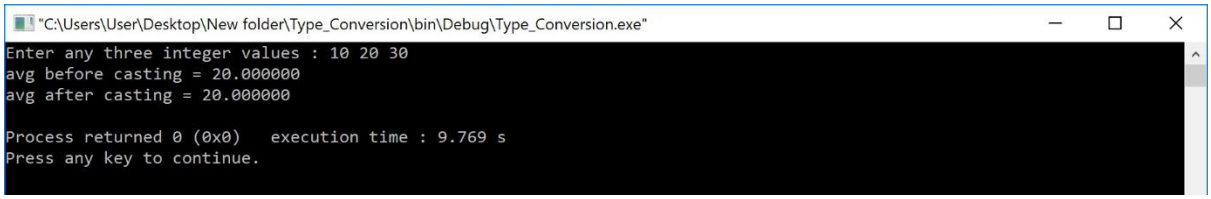
```
int totalMarks = 450, maxMarks = 600 ;
float average ;

average = (float) totalMarks / maxMarks * 100 ;
```

In the above example code, both totalMarks and maxMarks are integer data values. When we perform totalMarks / maxMarks the result is a float value, but the destination (average) datatype is a float. So we use type casting to convert totalMarks and maxMarks into float data type.

Example Program

```
#include<stdio.h>

#include<conio.h>


void main(){

   int a, b, c ;

   float avg ;

   cout << "Enter any three integer values : ";

   cin >> a >> b >> c;


   avg = (a + b + c) / 3 ;

   cout << "avg before casting = " << avg << endl;


   avg = (float)(a + b + c) / 3 ;

   cout << "avg after casting = " << avg << endl;


   return 0;

}
```

Output:



Bottom-Up Model and Top-Down Model

When designing complex systems, choosing the right approach for software development is important. Two fundamental design approaches are **Top-Down Design** and **Bottom-Up Design**. Each has its unique advantages, disadvantages, and use cases. In this article, we will explain the **Top-Down Design Model** and the **Bottom-Up Design Model**, highlighting their differences, benefits, and practical applications.

What is the Top-Down Design Model?

In the **top-down model**, an overview of the system is formulated without going into detail for any part of it. Each part of it then refined into more details, defining it in yet more details until the entire specification is detailed enough to validate the model. if we glance at a haul as a full, it's going to appear not possible as a result of it's so complicated For example: Writing a **University system program**, writing a word processor. Complicated issues may be resolved **victimization** high down style, conjointly referred to as Stepwise refinement where,

How Does Top-Down Approach Work?

The Top-Down approach works by:

1. Breaking the problem into major components,
2. Refining each component into smaller subcomponents
3. We Continue this process until every part is simple enough to implement.

This approach is particularly useful for solving complex problems, like developing a **University System Program** or a **Word Processor**. By starting with a high-level design and gradually focusing on smaller details, the complexity of the system becomes more manageable.

Advantages of the Top-Down Design Model

- **Simplifies Complex Problems:** Breaking problems into smaller parts help us to identify what needs to be done.
- **Easy to Identify Requirements:** At each step of refinement, new parts will become less complex and therefore easier to solve.
- **Promotes Reusability:** Parts of the solution may turn out to be reusable.
- **Collaboration-Friendly**: Breaking problems into parts allows more than one person to solve the problem.

What is Bottom-Up Design Model?

In contrast, the **Bottom-Up Design Model** is started by defining the system's individual parts first. Once the individual components are detailed, they are integrated into larger modules. This

process is continued till the system is fully integrated. The **Bottom-Up** approach is often used in Object-Oriented Programming (OOP) languages like **C++**, **Java**, and **Python**, where individual objects are identified and developed first.

How Does Bottom-Up Approach Work?

he Bottom-Up approach works by:

1. Identifying and specifying the **smaller components** (or objects).
2. Linking these smaller parts together to form larger components.
3. Continuously **integrating** them to complete the system.

This method focuses on creating well-defined and reusable low-level components before deciding how to integrate them into higher-level systems.

Advantages of the Bottom-Up Design Model

- **Reusability of Low-Level Components**: Decisions about reusable low-level utilities are made early in the design.
- **Focused Problem-Solving**: Developers can focus on solving smaller and more isolated problems first.
- **Increased Modularity**: The modular approach makes it easier to update individual components without affecting the entire system.

Key Differences Between Top-Down and Bottom-Up Design Models

S. No.	TOP DOWN APPROACH	BOTTOM UP APPROACH
1.	focus on breaking the problem into smaller, more manageable parts	Solves smaller problems and integrates them into a complete system
2.	Mainly used in Structured programming languages like COBOL, Fortran, C, etc.	Mainly used in Object Oriented Programming languages like C++, C#, Python.
3.	Each part is programmed separately therefore contains redundancy.	Redundancy is minimized by using data encapsulation and data hiding.
4.	communication is less among modules.	Modules must communicate to integrate the system.

S. No.	TOP DOWN APPROACH	BOTTOM UP APPROACH
5.	It is used for debugging and module documentation	It is basically used in testing.
6.	Decomposition of the system occurs, breaking it into smaller components.	Composition of the system happens by combining low-level components into a higher-level structure.
7.	The top function of system might be hard to identify.	In this sometimes we can not build a program from the piece we have started.
8.	implementation details can vary throughout the process	Building a program can be difficult if modules are not assembled in a logical order
9.	**Pros** • Easier isolation of interface errors • It benefits in the case error occurs towards the top of the program. • Defects in design get detected early and can be corrected as an early working module of the program is available.	**Pros** • Easy to create test conditions • Test results are easy to observe • It is suited if defects occur at the bottom of the program.
10.	**Cons-** • Difficulty in observing the output of test case. • Stub writing is quite crucial as it leads to setting of output parameters. • When stubs are located far from the top level module, choosing test cases and	**Cons-** • There is no representation of the working model once several modules have been constructed. • There is no existence of the program as an entity without the addition of the last module. • From a partially integrated system, test engineers cannot observe

S. No.	TOP DOWN APPROACH	BOTTOM UP APPROACH
	designing stubs become more challenging.	system-level functions. It can be possible only with the installation of the top-level test driver.

When to Use the Top-Down vs. Bottom-Up Design Models?

Use Top-Down Design Model When:

- We need to develop large and complex systems where **high-level architecture** is defined early.
- The overall system's structure is crucial before dealing with individual components.
- You are working in a more procedural or structured programming environment.

Use Bottom-Up Design Model When:

- You want to focus on individual components and gradually build them into a complete system.
- You are working with **Object-Oriented Programming** (OOP) languages and want to take advantage of modularity and object reusability.
- The system is composed of smaller, reusable components that need to be integrated.

Time Complexity and Space Complexity

Many times there are more than one ways to solve a problem with different algorithms and we need a way to compare multiple ways. Also, there are situations where we would like to know how much time and resources an algorithm might take when implemented. To measure performance of algorithms, we typically use time and space complexity analysis. The idea is to measure order of growths in terms of input size.

- Independent of the machine and its configuration, on which the algorithm is running on.
- Shows a direct correlation with the number of inputs.
- Can distinguish two algorithms clearly without ambiguity.

Time Complexity: The time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input. Note that the time to run is a function of the length of the input and not the actual execution time of the machine on which the algorithm is running on.

The valid algorithm takes a finite amount of time for execution. The time required by the algorithm to solve given problem is called ***time complexity*** of the algorithm. Time complexity is very useful measure in algorithm analysis.

Space Complexity:

Definition -

Problem-solving using computer requires memory to hold temporary data or final result while the program is in execution. The amount of memory required by the algorithm to solve given problem is called **space complexity** of the algorithm.

The space complexity of an algorithm quantifies the amount of space taken by an algorithm to run as a function of the length of the input. Consider an example: Suppose a problem to find the frequency of array elements.

It is the amount of memory needed for the completion of an algorithm.

To estimate the memory requirement we need to focus on two parts:

(1) A fixed part: It is independent of the input size. It includes memory for instructions (code), constants, variables, etc.

(2) A variable part: It is dependent on the input size. It includes memory for recursion stack, referenced variables, etc.

UNIT II

Control Structures Simple sequential programs Conditional Statements (if, if-else, switch), Loops (for, while, dowhile) Break and Continue.

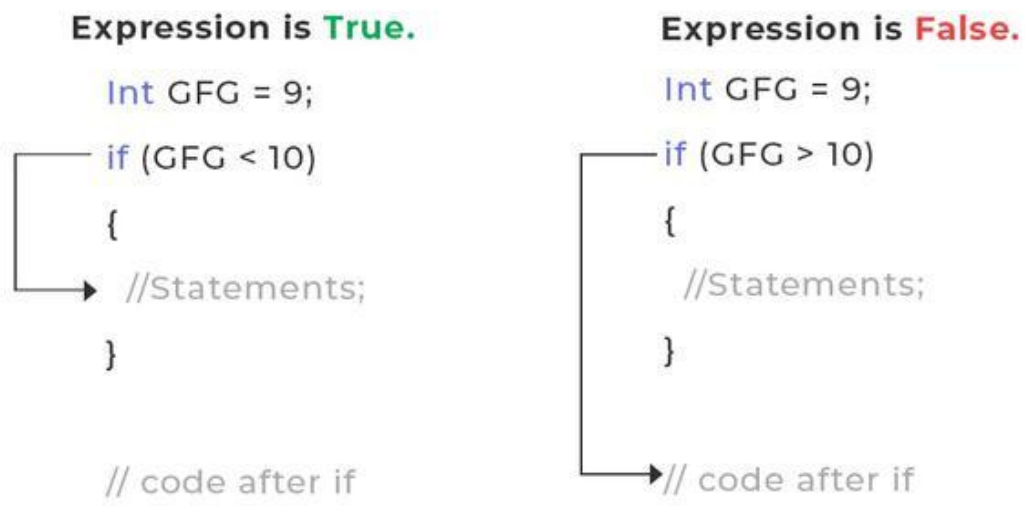
C - if Statement

The **if in C** is the simplest decision-making statement. It consists of the test condition and a block of code that is executed if and only if the given condition is true. Otherwise, it is skipped from execution.

Syntax of if in C

```
if (condition) {
// if body
// Statements to execute if condition is true
}
```

How if Statement works?

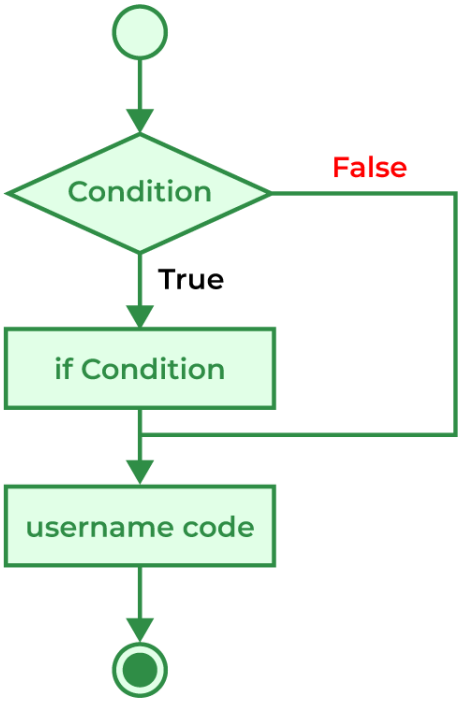


Working of if Statement in C

The working of the if statement in C is as follows:

1. **STEP 1:** When the program control comes to the if statement, the test expression is evaluated.
2. **STEP 2A:** If the condition is true, the statements inside the if block are executed.
3. **STEP 2B:** If the expression is false, the statements inside the if body are not executed.
4. **STEP 3:** Program control moves out of the if block and the code after the if block is executed.

Flowchart of if in C



Let's take a look at an example:

```
#include <stdio.h>

int main() {
  int n = 9;

  // if statement with true condition
  if (n < 10) {
    printf("%d < 10", n);
  }

  // if statement with false condition
  if (n > 20) {
    printf("%d > 20", n);
  }

  return 0;
```

```
}
```

Output

```
9 < 10
```

Explanation: The first if statement have a condition that checks if **n** is smaller than 10. For n = 9, this evaluated to true, so the code inside this if statement body is executed. In this second if statement, the condition checks if n is greater than 20, which in this case is evaluated to false. So, the code inside this if statement body is skipped.

Examples of if Statements

The below examples demonstrate the use of if statement to execute conditional code in a C program:

Check for Negative Number

We can check whether the number is negative by comparing it with 0 and checking if it is less than zero, it is negative.

```
#include <stdio.h>


int main() {
  int n = -4;


  // Condition to check for negative number
  if (n < 0) {
    printf("%d is Negative", n);
  }


  return 0;
}
```

Output

```
-4 is Negative
```

Check Whether the Number is Even or Odd

In this program, we will make use of the logic that if the number is divisible by 2, then it is even else odd.

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

```
int main() {
  int n = 4;

  // Condition to check for even number
  if (n % 2 == 0) {
    printf("%d is Even", n);
  }

  // Condition to check for odd number
  if (n % 2 != 0) {
    printf("%d is Odd", n);
  }

  return 0;
}
```

Output

4 is Even

C if else Statement

The **if else** in C is an extension of the if statement which not only allows the program to execute one block of code if a condition is true, but also a different block if the condition is false. This enables making decisions with two possible outcomes.

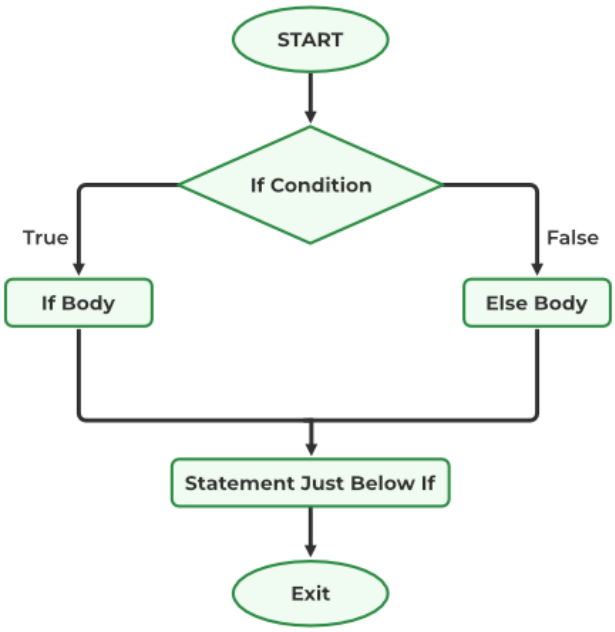
Syntax of if-else Statement

```
if (condition) {
  // Code to execute if condition is true
}
else {
  // Code to execute if condition is false
}
```

If the **condition** is **true**, then the code inside the **if** block is executed, otherwise the code inside the **else** block is executed. Any non-zero and non-null values are assumed to be true, and zero or null values are assumed to be false.

Working of if-else statement

The below flowchart explains the if else works in C:



Flowchart of if-else in C

- The **if-else statement** works by checking the condition defined with the if statement.
- If the condition defined in the if statement is true, then the code inside the if block is executed and the rest is skipped.
- If the condition evaluates to false, then the program executes the code in the else block

Examples of if-else

The below examples illustrate how to use the if else statement in C programs

Negative Number check using If-else statement

```
#include <stdio.h>


int main() {
  int n = -7;


  // If the number is negative
  if (n < 0)
    printf("Negative");


  // If the number is not negative
  else
```

```
    printf("Not Negative");

  return 0;
}
```

Output

Negative

Explanation: In this program, the if statements check if the given number **n** is less than 0 which means that the number is negative. As **n** is negative, it prints the if block. But if the number **n** was positive, the else block would have been executed.

Note: *If long at the block only contains the single statement, we can skip the curly braces.*

Check if Integer Lies in the Range

```
#include <stdio.h>

int main() {
  int n = 6;

  // Check if the number lies in the range [10, 20]
  if (n >= 10 && n <= 20) {
    printf("%d lies in range.", n);
  }
  else {
    printf("%d does not lie in range.", n);
  }

  return 0;
}
```

Largest Among Three Numbers Using If-else

```
#include <stdio.h>

int main() {
  int a = 1, b = 2, c = 11;
```

```
  // Finding the largest by comparing using
  // relational operators with if-else
  if (a >= b) {
    if (a >= c)
      printf("%d", a);
    else
      printf("%d", c);
  }
  else {
    if (b >= c)
      printf("%d", b);
    else
      printf("%d", c);
  }

  return 0;
}
```

C if , else if ladder

In C, **if else if ladder** is an extension of if else statement used to test a series of conditions sequentially, executing the code for the first true condition. A condition is checked only if all previous ones are false. Once a condition is true, its code block executes, and the ladder ends.

Component of if, else if ladder

- **if statement:** This is the first statement in the if else if ladder. The if statement contains a condition and a code block which is executed if the condition is evaluates to true else the control is passed to the nest statement.

Syntax

```
if(condition)
{
  code to be executed
}
```

- **else if statement:** The 'else if' statement always comes after the if statement. An if else if ladder can have multiple else if statements, where each have a condition that is to be

tested along with a code block which will be executed if the condition evaluates to true else the control is passed to the nest statement. A particular else if statements is executed only when the conditions mentioned in the **'if'** and '**else if'** statements before the current **'else if'** evaluates to false.

Syntax

if (condition){.....}

else if(condition)

{

code to be executed

}

- **else statement:** This is the last statement in the if else if ladder and it is optional. The else statement does not contains any condition and only contain a code block which is executed if all the conditions mentioned in the above **'if'** and '**else if'** statements evaluates to false.

Syntax

if (condition) {....}

else if (condition) {....}

else {

code to be executed.

}

Syntax of Complete if, else if Ladder

if (condition1) {

// Statements 1

}

else if (condition2) {

// Statements 2

}

else {

// Else body

}

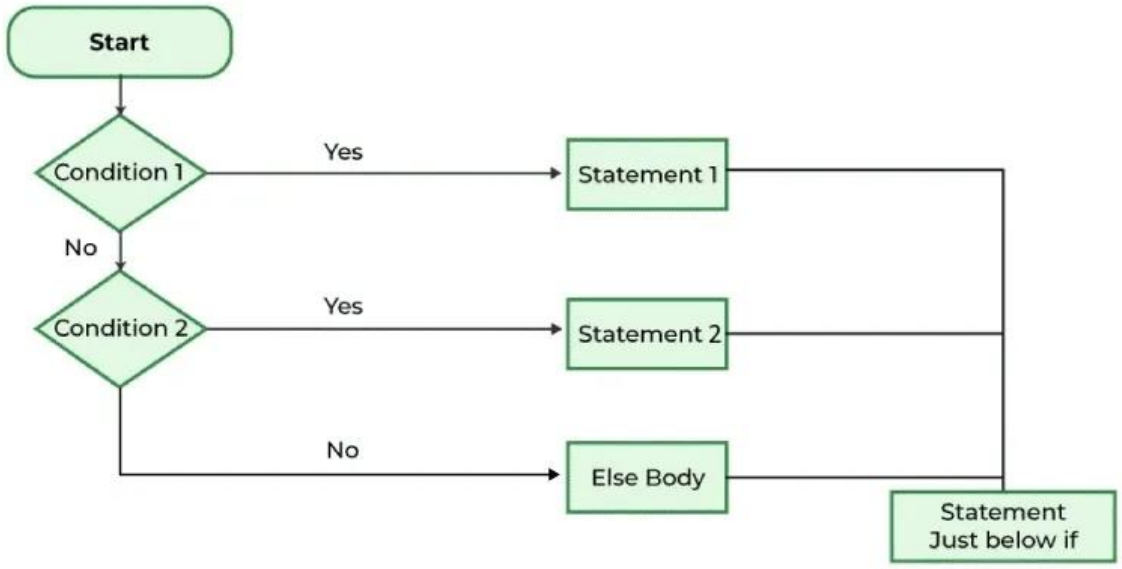
where,

- **condition1, condition2**: Contitions that are to be tested.
- **statements 1, statements 2**: Code block corresponding to each condition.

- **else body**: code block to be executed if none of the condition evaluate to true.

Working of if, else if Ladder

The working of if else if ladder can be understood using the following flowchart:



- If **Condition 1** evaluates to true, **Statement 1** is executed, and the rest of the else if and else conditions are skipped.
- If **Condition 1** is false, **Condition 2** is evaluated.
- If **Condition 2** evaluates to true, **Statement 2** is executed, and the else block is skipped.
- If **Condition 2** is also false, the **Else Body** is executed.
- After completing the if-else-if ladder, the **statement just below the if** are executed.

Examples

The following examples demonstrate the use of if else if ladder in different coding problems:

Check if a number is positive, negative or 0

```
#include <stdio.h>


int main() {
   int n = 10;
   printf("Number = %d \n", n);
   // Check if the number is positive, negative, or zero
   if (n > 0) {
      printf("Positive.\n");
```

```
    } else if (n < 0) {
        printf("Negative.\n");
    } else {
        printf("Zero.\n");
    }

    return 0;
}
```

Output

Number = 10

Positive.

Explanation: The if-else if ladder checks whether the number is greater than zero (positive), less than zero (negative), or exactly zero. Based on the result, the corresponding message is printed.

Switch Statement in C

C switch statement is a conditional statement that allows you to execute different code blocks based on the value of a variable or an expression. It is often used in place of if-else ladder when there are multiple conditions.

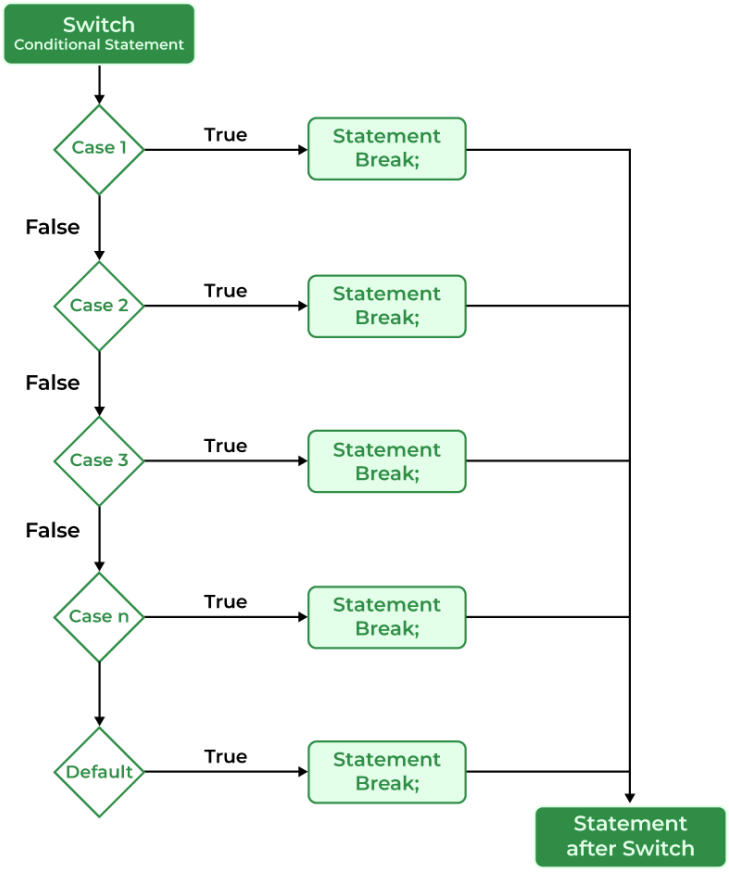
Syntax of switch

```
switch (expression) {
    case value1:
        // Code_block1
        break;

    case value2:
        // Code_block2
        break;

    default:
        // Default_code_block
}
```

Here,

- **expression**: Expression or variable whose value is tested. It should **result in an integer or character value.**
- **case**: Each switch case represents a block of code that will be executed with the expression evaluates to its corresponding value. There should be atleast one case in the switch.
- **value1**, **value2**: These are the possible values of the expression for which the corresponding cases are executed. These values must be **unique.**
- **break**: This is used to **exit the switch** statement once a case is executed. Without break, the program will continue executing the subsequent cases (this is called "fall through").
- **default**: This block of code is executed if **none of the case values match** the expression. It is optional to add this block.

Flowchart of C switch Statement



Working of Switch Statement

The working of the switch statement in C is as follows:

- ***Step 1:*** *The switch variable is evaluated*
- ***Step 2:*** *The evaluated value is matched against all the present cases.*
- ***Step 3A:*** *If the matching case value is found, the associated code is executed.*

- ***Step 3B:*** *If the matching code is not found, then the default case is executed if present.*
- ***Step 4A:*** *If the break keyword is present in the executed case, then program control breaks out of the switch statement after executing the code block associated with the first matching condition.*
- ***Step 4B:*** *If the break keyword is not present, then all the cases after the matching case are executed*
- ***Step 5:*** *Statements after the switch statement are executed.*

Examples of Switch Statement

The below programs show some use cases of switch statement in practical scenario:

Print Day Name of the Week

```
#include <stdio.h>


int main() {


  // Day number
  int day = 2;


  // Switch statement to print the name of the day
  // on the basis of day number
  switch (day) {
  case 1:
    printf("Monday");
    break;
  case 2:
    printf("Tuesday");
    break;
  case 3:
    printf("Wednesday");
    break;
  case 4:
    printf("Thursday");
    break;
```

```
    case 5:
      printf("Friday");
      break;
    case 6:
      printf("Saturday");
      break;
    case 7:
      printf("Sunday");
      break;
    default:
      printf("Invalid Input");
      break;
    }

    return 0;
}
```

Output

Tuesday

Simple Calculator using Switch Statement

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

int main() {
    // two number
    int x, y;
    // Variable that select operation to perform
    // i.e. switch variable
    char choice;

    // Take input
```

```
    printf("Enter the Operator (+,-,*,/)\n");
    scanf(" %c", &choice);

    printf("Enter the two numbers: ");
    scanf("%d %d", &x, &y);

    // switch case with operation for each operator
    switch (choice) {
    case '+':
        printf("%d + %d = %d\n", x, y, x + y);
        break;

    case '-':
        printf("%d - %d = %d\n", x, y, x - y);
        break;

    case '*':
        printf("%d * %d = %d\n", x, y, x * y);
        break;
    case '/':
        printf("%d / %d = %d\n", x, y, x / y);
        break;
    default:
        printf("Invalid Operator Input\n");
    }

    return 0;
}
```

Output

```
Enter the Operator (+,-,*,/)
+
```

Enter the two numbers: 10 20
10 + 20 = 30

C switch Statement without Break

The **break** keyword is used to stop the execution inside a switch block. It helps to terminate the switch block and break out of it. If omitted, execution will continue on into the next case.

```
#include <stdio.h>


int main() {
  int var = 2;


  // switch case without break
  switch (var) {
  case 1:
    printf("Case 1 is executed.\n");
  case 2:
    printf("Case 2 is executed.\n");
  case 3:
    printf("Case 3 is executed.\n");
  case 4:
    printf("Case 4 is executed.");
  }
  return 0;
}
```

Output

Case 2 is executed.

Case 3 is executed.

Case 4 is executed.

C - Loops

In C programming, there is often a need for repeating the same part of the code multiple times. For example, to print a text three times, we have to use printf() three times as shown in the code:

```
#include <stdio.h>


int main() {
   printf( "Hello GfG\n");
   printf( "Hello GfG\n");
   printf( "Hello GfG");
   return 0;
}
```

Output

```
Hello GfG
Hello GfG
Hello GfG
```

But if we say to write this 20 times, it will take some time to write statement. Now imagine writing it 100 or 1000 times. Then it becomes a really hectic task to write same statements again and again. To solve such kind of problems, we have loops in programming languages.

```
#include <stdio.h>


int main() {


   // Loop to print "Hello GfG" 3 times
   for (int i = 0; i < 3; i++) {
      printf("Hello GfG\n");
   }
   return 0;
}
```

Output

```
Hello GfG
Hello GfG
```

Hello GfG

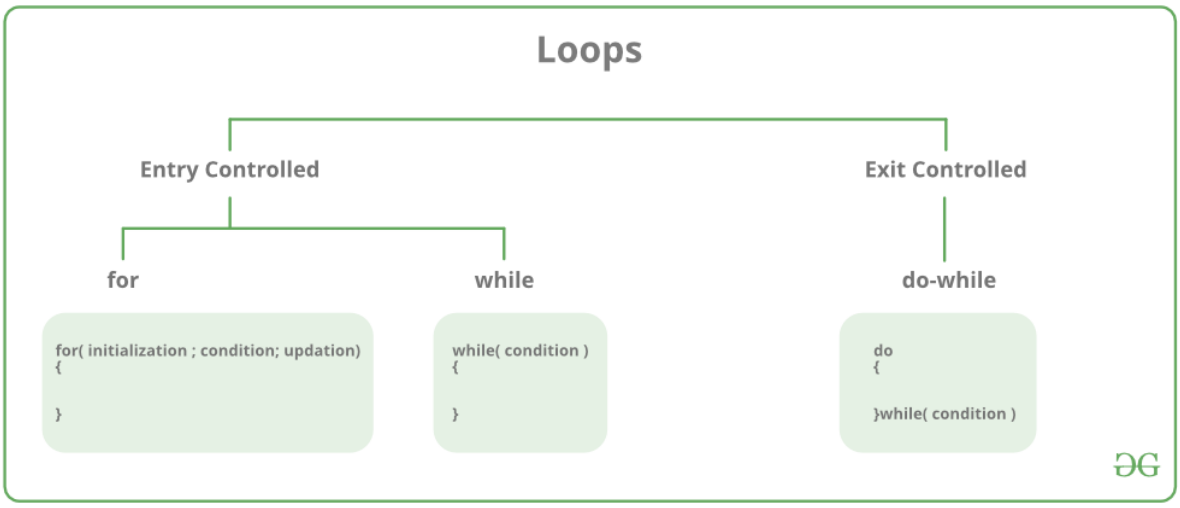
What are loops in C?

Loops in C programming are used to repeat a block of code until the specified condition is met. It allows programmers to execute a statement or group of statements multiple times without writing the code again and again.

Types of Loops in C

There are 3 looping statements in C:

Table of Content



C for Loop

In C programming, the 'for' loop is a control flow statement that is used to repeatedly execute a block of code as many times as instructed. It uses a variable (loop variable) whose value is used to decide the number of repetitions. It is commonly used to iterate over a sequence such as an array or list.

C for Loop Syntax

for (initialization; condition; updation) {

// Body of the loop

}

where,

- **Initialization**: This step is executed first, and it is typically used to initialize a loop control variable.

- **Condition**: This conditional expression is evaluated before each iteration. and if the condition is true, the loop body executes. if it's false, the loop terminates.
- **Updation**: This step is used to update the loop control variable and is executed after each iteration.
- **body:** The body of the loop are the set of statements that are executed when the condition is true.

Note: *As we can see, unlike the while loop and do...while loop, the for loop contains the initialization, condition, and updating statements for loop as part of its syntax.*

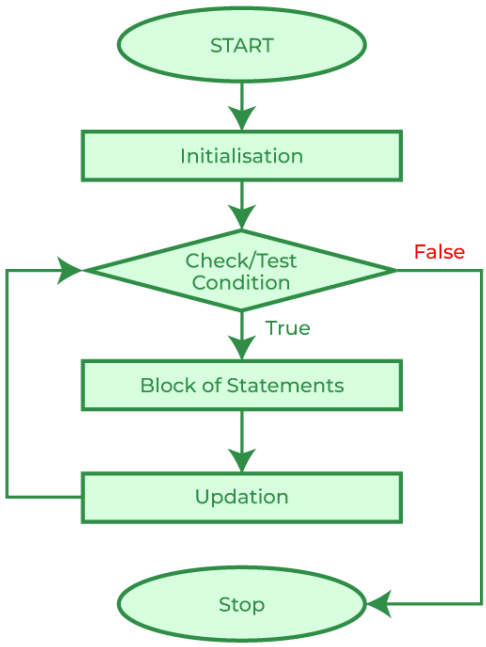
Working of For Loop

The working of for loop is mentioned below:

- ***Step 1:*** *Initialization is the basic step of for loop this step occurs only once during the start of the loop. During Initialization, variables are declared, or already existing variables are assigned some value.*
- ***Step 2:*** *During the Second Step condition statements are checked and only if the condition is the satisfied loop we can further process otherwise loop is broken.*
- ***Step 3:*** *All the statements inside the loop are executed.*
- ***Step 4:*** *Updating the values of variables has been done as defined in the loop.* ***Continue to Step 2 till the loop breaks.***

Flowchart of for Loop

The following flow chart defines the logic behind the working of for loop.



C For Loop Examples

The below examples demonstrate the use of for loop in different cases:

First N Natural Numbers Using For Loop

```
#include <stdio.h>

int main() {
   int n = 5;

   // Initialization of loop variable
   int i;
   for (i = 1; i <= n; i++)
      printf("%d ", i);

   return 0;
}
```

Output

1 2 3 4 5

Print Table from 1 to 5 Using For Loop

```
#include <stdio.h>

int main()  {

   // Outer for loop to print a multiplication
   // table for all numbers upto 5
   for (int i = 1; i <= 5; i++) {

      // Inner loop to print each value in table
      for (int j = 1; j <= 5; j++) {
         printf("%d ", i * j);
      }
```

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

Output

1 2 3 4 5

2 4 6 8 10

3 6 9 12 15

4 8 12 16 20

5 10 15 20 25

Explanation: This code uses nested for loops to generate a multiplication table up to 5. The outer loop iterates through the rows (from 1 to 5), and the inner loop (controlled by j) iterates through the columns (also from 1 to 5). For each combination of i and j, the product i * j is printed, creating the table entries.

Infinite For Loop

This is a kind of *for loop* where the input parameters are not available or the condition that's always true due to which, the loop iterates/runs endlessly.

```
#include <stdio.h>

int main() {

    // for loop with no initalization, condition
    // and updation
    for (;;) {
        printf("GfG\n");
    }

    return 0;
}
```

Output

GfG
GfG
.
.
.
infinity

while Loop in C

The **while loop** in C allows a block of code to be executed repeatedly as long as a given condition remains true. It is often used when we want to repeat a block of code till some condition is satisfied.

Syntax of while Loop

while (condition) {

// Body

updation

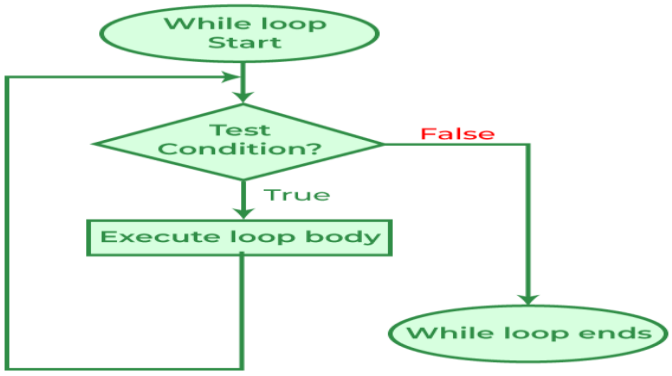
}

where,

- **Initialization:** In this part, we initialize the **loop variable** to some **initial value.** Initialization is not part of while loop syntax but it is essential when we are using some variable in the test expression
- **Conditional:** This is one of the most crucial part as it decides whether the block in the while loop code will execute. The while loop body will be executed if and only the **test condition** defined in the conditional statement is **true.**
- **Body:** It is the actual set of statements that will be executed till the specified condition is true.
- **Updation:** It is an expression that **updates** the value of the **loop variable** in each iteration. It is also not part of the syntax, but we have to define it explicitly in the body of the loop.

Working of while Loop

Let's understand the working of while loop in C using the flowchart given below:



We can understand the working of the while loop by looking at the above flowchart:

- **STEP 1:** When the program first comes to the loop, the test condition will be evaluated.
- **STEP 2A:** If the test condition is **false,** the body of the loop will be skipped program will continue.
- **STEP 2B:** If the expression evaluates to true, the body of the loop will be executed.
- **STEP 3:** After executing the body, the program control will go to STEP 1. This process will continue till the test expression is true.

Examples of while Loop

The below examples show how to use a while loop in a C program:

Sum of First N Natural Numbers using While loop

```
#include <stdio.h>


int main() {
   int sum = 0, i = 1;


   while (i <= 10) {


      // Add the current value of i to sum
      sum += i;


      // Increment i
      i++;
   }


   printf("%d", sum);
   return 0;
}
```

Output

55

Multiplication Table from 1 to 5

```
#include <stdio.h>

int main() {

  // Initialize outer loop counter
  int i = 1;

  // Outer while loop to print a multiplication
  // table for all numbers up to 5
  while (i <= 5) {

    // Initialize inner loop counter for each row
    int j = 1;

    // Inner while loop to print each value in table
    while (j <= 5) {
      printf("%d ", i * j);
      j++;
    }
    printf("\n");
    i++;
  }
  return 0;
}
```

Output

1 2 3 4 5

2 4 6 8 10

3 6 9 12 15

4 8 12 16 20

5 10 15 20 25

Explanation: In the above program, one while loop is nested inside anther while loop to print each value in the table. This is called nesting of loops and why can nest as many while loops as we want in in C.

Infinite while loop

An infinite while loop is created when the given condition always remains true. It is generally encountered in programs where, the condition variable is not correctly updated or has some logic error.

```
#include <stdio.h>


int main() {


  // Infinite loop condition (1 is always true)

  while (1) {

    printf("GfG\n");

  }


  // This line is never reached due to infinite loop

  return 0;

}
```

Output

```
GfG
GfG
.
.
infinite
```

As seen in the above example, the loop will continue till infinite because the loop variable will always remain the same resulting in the condition that is always true.

Important Points about while loop

- It is an entry-controlled loop.
- It runs the block of statements till the condition is satisfied, once the condition is not satisfied it will terminate.
- Its workflow is ***firstly it checks the condition and then executes the body. Hence, a type of pre-tested loop.***
- This loop is generally preferred over for loop when the number of iterations is unknown.

do...while Loop in C

C do...while loop is a type of loop that executes a code block until the given condition is satisfied. Unlike the while loop, which checks the condition before executing the loop, the do...while loop checks the condition after executing the code block, ensuring that the code inside the loop is executed at least once, even if the condition is false from the start.

Syntax

do {

// Body of the loop

// Update expression

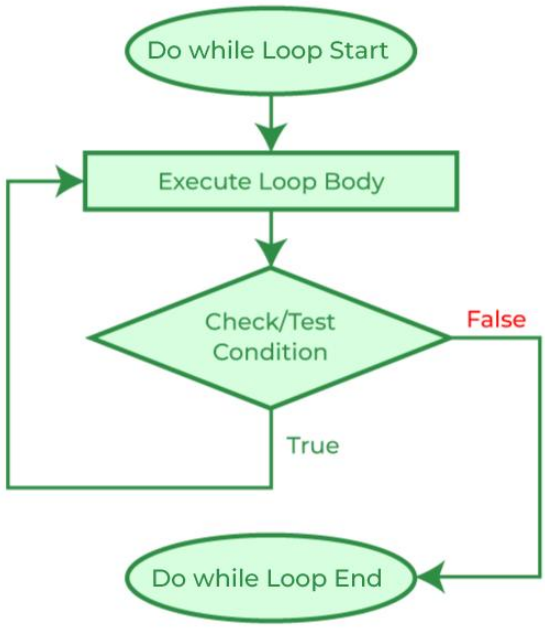
} **while** (condition);

where,

- **condition:** The loop condition which will decide the number of times the loop runs.
- **update expression:** The expression that updates the condition to move it towards the loop end in each repetition.

Working of do...while Loop

Let's understand the working of do while loop using the below flowchart.



Flowchart of do...while Loop in C

1. When the program control comes to the do...while loop, **the body of the loop is executed first and then the test condition/expression is checked**, unlike other loops where the test condition is checked first. Due to this property, the do...while loop is also called exit controlled or post-tested loop.

2. When the test condition is evaluated as **true**, the **program control goes to the start** of the loop and the body is executed once more.
3. The above process repeats till the test condition is true.
4. When the test condition is evaluated as **false, the program controls move on to the next statements** after the do...while loop.

The initialization and updation is not a part of the do...while loop syntax. We have to do that explicitly before and in the loop respectively.

Examples of do while Loop

The below programs demonstrate how to use the do while loops in C programs in different situations:

do...while Loop for False Condition

```
#include <stdbool.h>

#include <stdio.h>


int main() {


  // Declaring a false variable

  bool c = false;


  do {

    printf("This is loop body.");


   // False condition

  } while (c);


  return 0;

}
```

Output

This is loop body.

Explanation: As we can see, the body of the loop is executed even if the condition was false from the start.

Print Matrix using do-while loop

We can also nest one do...while loop into another loop (called nested loops). This helps in printing multidimensional structures such as matrices.

```
#include <stdio.h>

int main() {

  // Declaring loop variables
  int i = 0, j;
  int c = 0;

  // Outer loop starts
  do {
    j = 0;

    // inner loop starts
    do {
      printf("%d  ", c++);
      j++;
    } while (j < 3);
    printf("\n");
    i++;
  } while (i < 3);

  return 0;
}
```

Output

```
0  1  2
3  4  5
6  7  8
```

Difference between while and do...while Loop

The following table lists the important **differences between the while and do...while Loop** .

while Loop	do...while Loop
The test condition is checked **before the loop body is executed.**	The test condition is checked **after executing the body.**
When the condition is false, the **body is not executed** not even once.	The body of the **do...while loop is executed at least once** even when the condition is false.
It is a type of **pre-tested or entry-controlled loop.**	It is a type of **post-tested or exit-controlled loop.**
Semicolon is not required.	Semicolon is required at the end.

Continue Statement in C

The **continue statement** in C is a jump statement used to skip the current iteration of a loop and continue with the next iteration. It is used inside loops (for, while, or do-while) along with the conditional statements to bypass the remaining statements in the current iteration and move on to the next iteration.

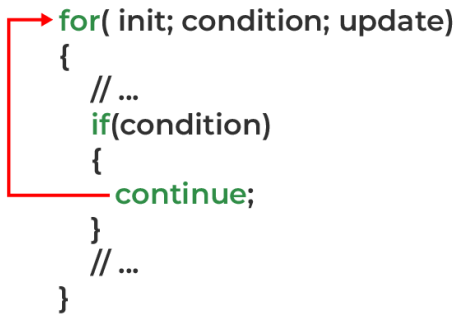
Syntax of Continue in C

The syntax of continue is just the **continue** keyword placed wherever we want in the loop body.

continue;

The continue statement is used with conditional statements such as **if-else** to specify the condition for when to skip the iteration.

Working of C Continue



Working of C continue in for Loop

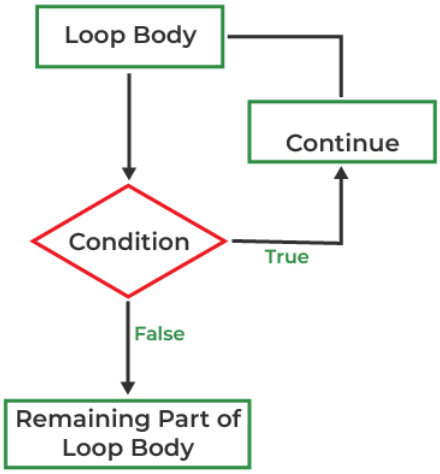
Working of the Continue Statement:

- The loop's execution starts after the loop condition is evaluated to be true.

- The **condition** of the continue statement will be evaluated.
    - If the condition is **false**, the normal execution will continue.
    - If the condition is **true**, the program control will jump to the start of the loop and all the statements below the continue will be skipped.
- Steps 1 and 2 will be repeated till the end of the loop.

Flowchart of Continue Statement

The flowchart of the continue can be constructed using the understanding of the continue we got from above.



Flowchart of the continue Statement in C

Examples of C continue

The below program illustrates how to use continue statements in our C programs

Print Odd Numbers from 1 to 10

```
#include <stdio.h>


int main() {
  int n = 1;


  // Loop from 1 to 10
  while (n <= 10) {


    // If the number is even, skip iteration
    // to avoid printing it
```

```
        if (n % 2 == 0) {
            n++;
            continue;
        }
        printf("%d ", n);
        n++;
    }

    return 0;
}
```

Output

1 3 5 7 9

Explanation: The condition checks if the number **n** is even and when it's true it skips to the next iteration which prevents the print statement from executing as the print statement is below the continue so when the continue is executed (for even numbers), no printing is done.

break vs continue

The primary difference between the break and continue statements is that by using **break statement**, we terminate the smallest enclosing loop (e.g, a while, do-while, for, or switch statement). On the other hand, by using the **continue statement,** we only skip the current iteration of the loop and execure the next iteration.

Break Statement in C

The break statement in C is a loop control statement that breaks out of the loop when encountered. It can be used inside loops or switch statements to bring the control out of the block. The break statement can only break out of a single loop at a time.

Syntax of break in C

```
// in a block {
    break;
}
```

We just put the break wherever we want to terminate the execution of the loop.

How break in C Works?

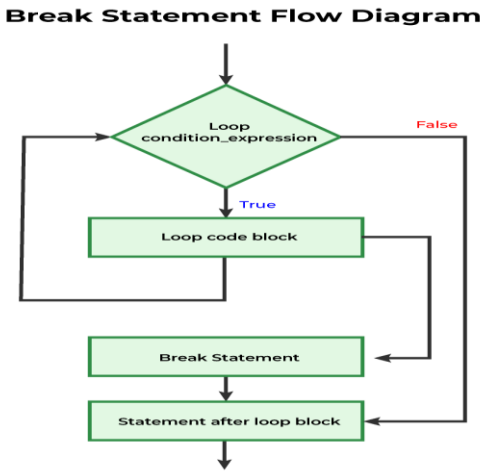
```
for( init; condition; operation)
{
   // code
   if(condition to break)
   {
      break;
   }
   // code
}
```

Working of break stateemnt in C

The working of the break statement in C is described below:

1. ***STEP 1:*** *The loop execution starts after the test condition is evaluated.*
2. ***STEP 2:*** *If the break condition is present the condition will be evaluated.*
3. ***STEP 3A****: If the condition is true, the program control reaches the break statement and skips the further execution of the loop by jumping to the statements directly below the loop.*
4. ***STEP 3B****: If the condition is false, the normal flow of the program control continues.*

Flowchart of break Statement



Examples of break in C

The following examples illustrate the use of break in C programming:

break with Nested Loops

```
#include <stdio.h>


int main() {
```

```
    // Nested for loops with break statement
    // at inner loop
    for (int i = 1; i <= 6; ++i) {
        for (int j = 1; j <= i; ++j) {
            if (i <= 4) {
                printf("%d ", j);
            }
            else {

                // If i > 4 then this innermost loop will
                // break
                break;
            }
        }
        printf("\n");
    }
    return 0;
}
```

Output

1

1 2

1 2 3

1 2 3 4

Explanation: In the above program the inner loop breaks when the value of i becomes equal to 4, which stops the printing of the value , although the outer loop continues to run but as the inner loop encounters a break statement for every iteration of i after the value of i becomes equal to 4, no values are printed after the fourth line.

goto Statement in C

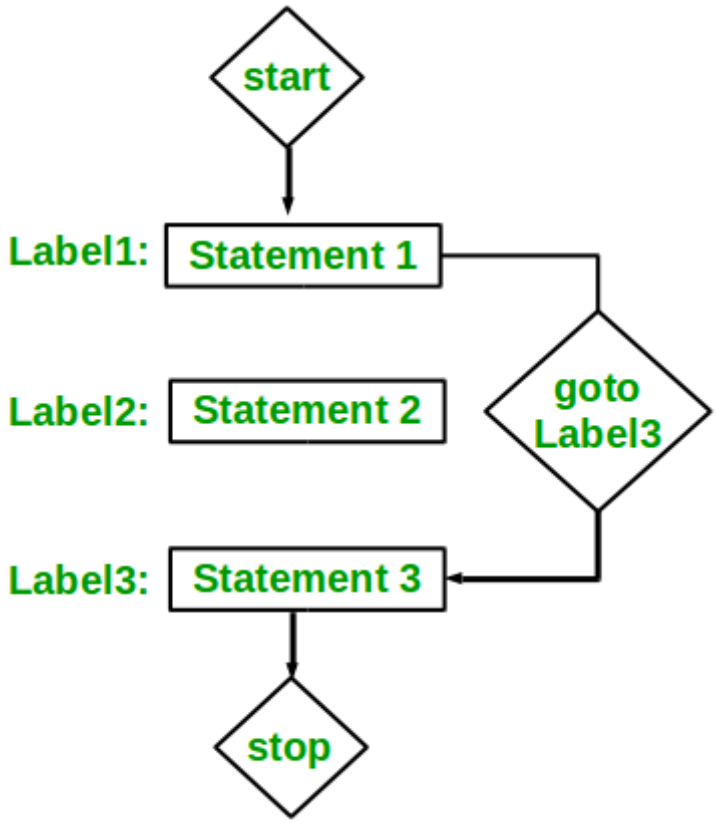
The **goto statement** in C allows the program to jump to some part of the code, giving you more control over its execution. While it can be useful in certain situations, like error handling or exiting complex loops, it's generally not recommended because it can make the code harder to read and maintain.

Syntax

Syntax1 | *Syntax2*
---
goto label; | *label:*
. | .
. | .
. | .
label: | *goto label;*

In the above syntax, the first line tells the compiler to go to or jump to the statement marked as a label. Here, the label is a user-defined identifier that indicates the target statement. The statement immediately followed after 'label:' is the destination statement. The 'label:' can also appear before the 'goto label;' statement in the above syntax.

Flowchart of goto



Flowchart of goto Statement in C

The goto statement allows for jumping to specific parts of a program and understanding its function is still valuable.

Examples of goto

Below are some examples of how to use a goto statement.

Check Even or Odd Number

#include <stdio.h>

```
int main() {
   int n = 26;

   if (n % 2 == 0)

      // jump to even
      goto even;
   else

      // Jump to odd
      goto odd;

even:
   printf("%d is even", n);
   return 0;

odd:
   printf("%d is odd", n);
   return 0;
}
```

Output

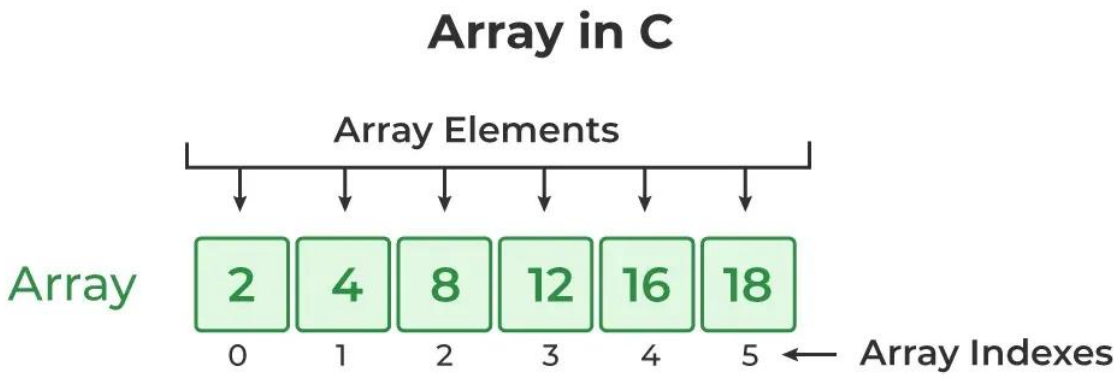
26 is even

UNIT III

Arrays and Strings Arrays indexing, memory model, programs with array of integers, two dimensional arrays, Introduction to Strings.

C Arrays

An array in C is a fixed-size collection of similar data items stored in contiguous memory locations. It can be used to store the collection of primitive data types such as int, char, float, etc., as well as derived and user-defined data types such as pointers, structures, etc.



Creating an Array in C

The whole process of creating an array in C language can be divided into two primary sub processes i.e.

1. Array Declaration

Array declaration is the process of specifying the type, name, and size of the array. In C, we have to declare the array like any other variable before using it.

data_type array_name[size];

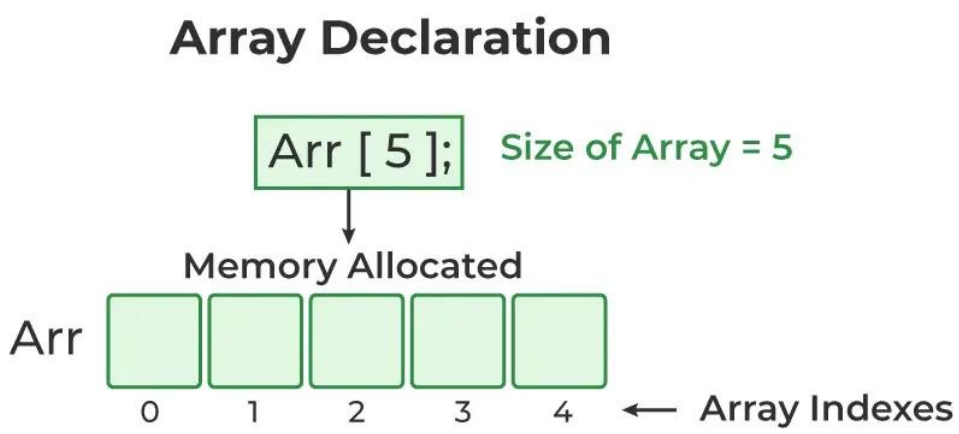
The above statements create an array with the name **array_name**, and it can store a specified number of elements of the same data type.

Example:

// Creates array arr to store 5 integer values.

int arr[5];

When we declare an array in C, the compiler allocates the memory block of the specified size to the array name.



2. Array Initialization

Initialization in C is the process to assign some initial value to the variable. When the array is declared or allocated memory, the elements of the array contain some garbage value. So, we need to initialize the array to some meaningful values.

Syntax:

int arr[5] = {2, 4, 8, 12, 16};

The above statement creates an array **arr** and assigns the values **{2, 4, 8, 12, 16}** at the time of declaration.

We can skip mentioning the size of the array if declaration and initialisation are done at the same time. This will create an array of size n where n is the number of elements defined during array initialisation. We can also **partially initialize** the array. In this case, the remaining elements will be assigned the value 0 (or equivalent according to the type).

//Partial Initialisation

int arr[5] = {2, 4, 8};

//Skiping the size of the array.

int arr[] = {2, 4, 8, 12, 16};

//initialize an array with all elements set to 0.

int arr[5] = {0};

1 / 3

Accessing Array Elements

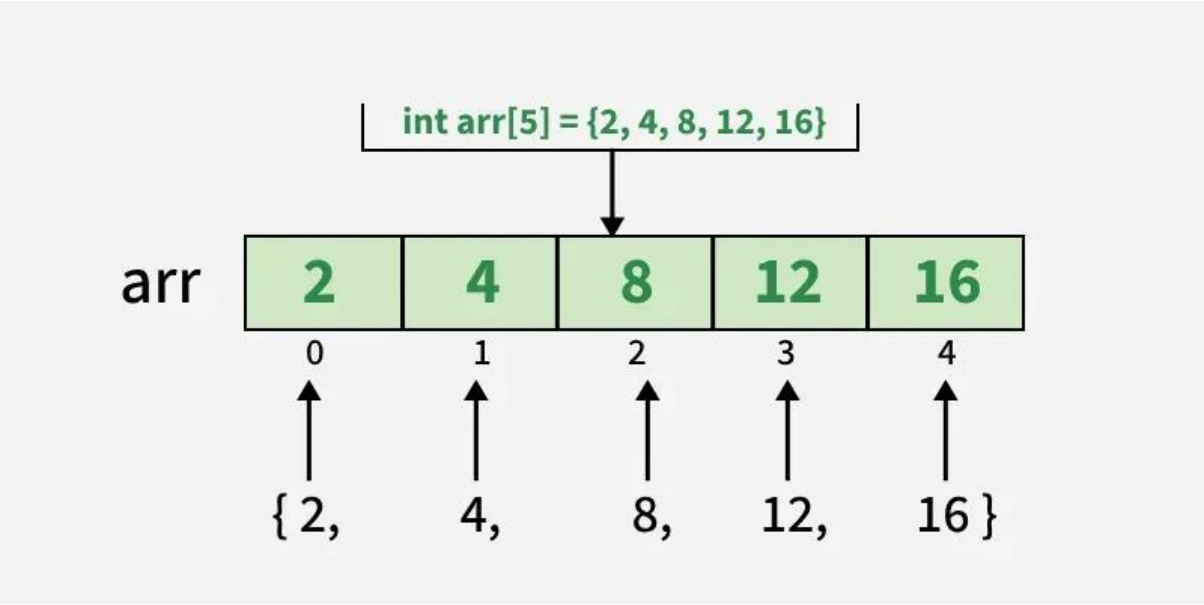
Array in C provides random access to its elements, which means that we can access any element of the array by providing the position of the element, called the index.

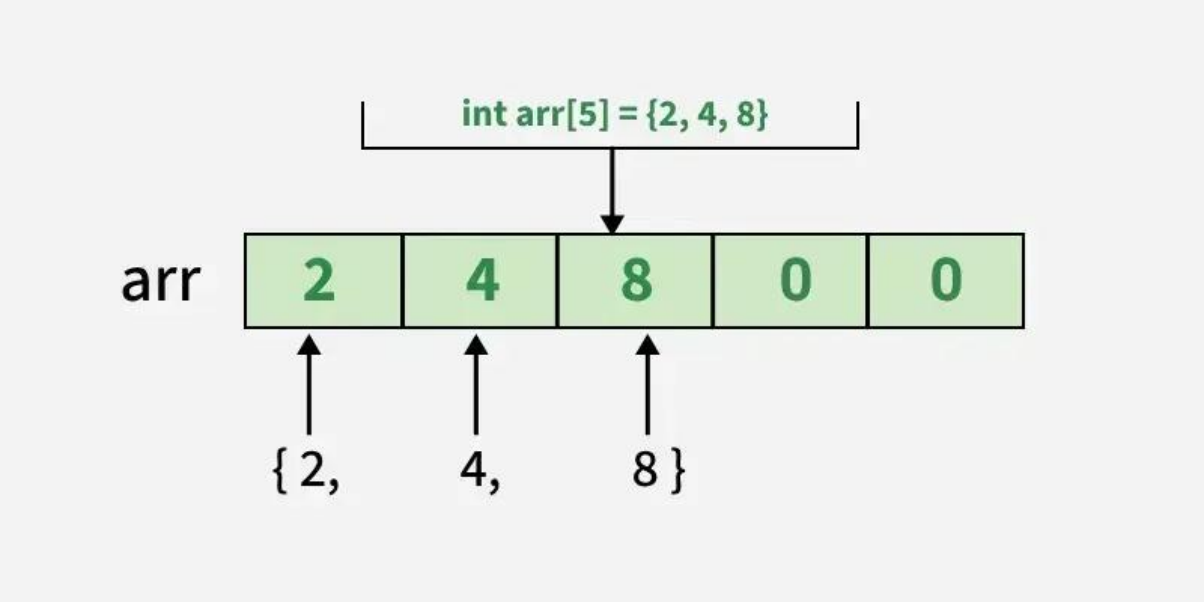
Syntax:

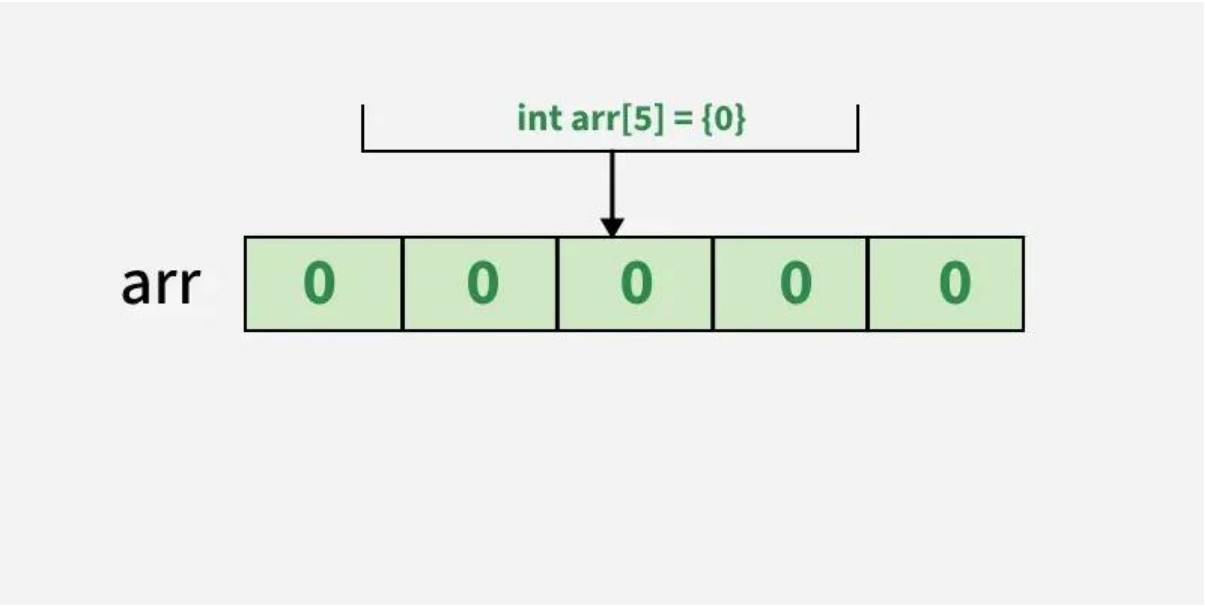
The index values start from **0** and goes up to **array_size-1**. We pass the index inside **square brackets []** with the name of the array.

array_name [index];

where, index value lies into this range - **(0 ≤ index ≤ size-1)**.





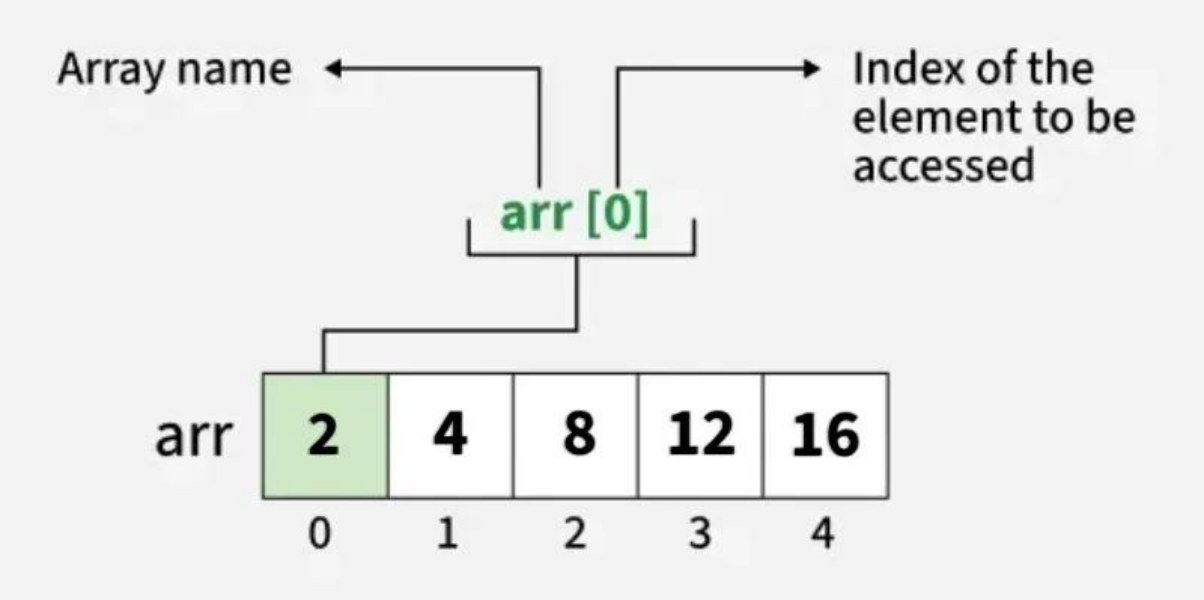


Example:

```
#include <stdio.h>

int main() {

   // array declaration and initialization
   int arr[5] = {2, 4, 8, 12, 16};

   // accessing element at index 2 i.e 3rd element
   printf("%d ", arr[2]);

   // accessing element at index 4 i.e last element
   printf("%d ", arr[4]);

   // accessing element at index 0 i.e first element
   printf("%d ", arr[0]);
   return 0;
}
```

Output

8 16 2



Update Array Element

We can update the value of array elements at the given index i in a similar way to accessing an element by using the array **square brackets []** and **assignment operator (=)**.

array_name[i] = new_value;

Example:

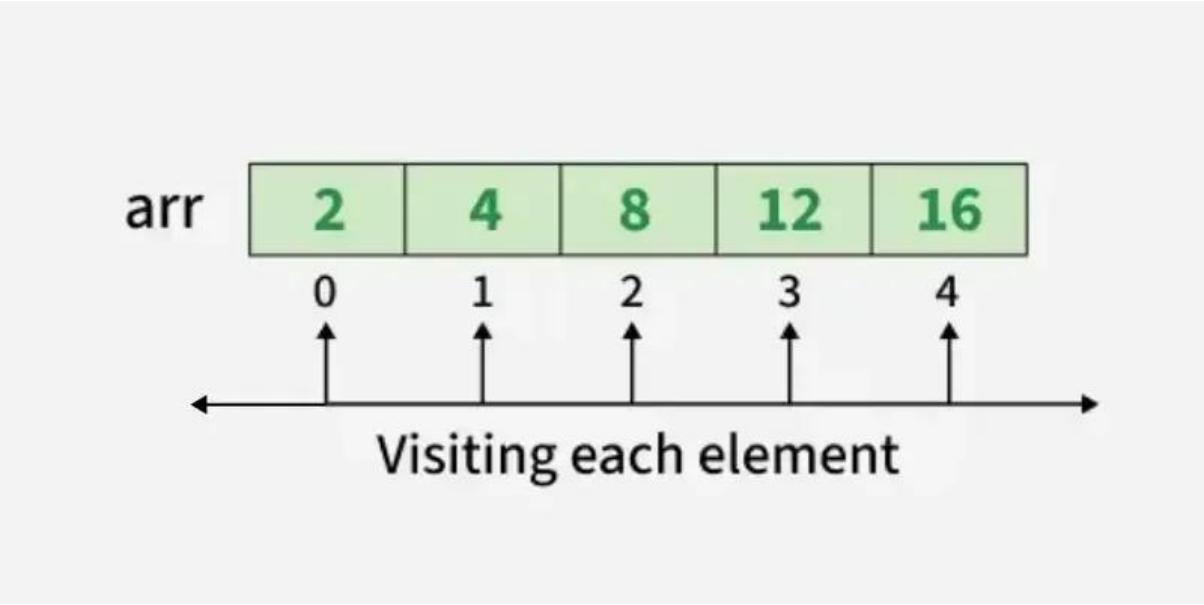
```
#include <stdio.h>


int main() {
   int arr[5] = {2, 4, 8, 12, 16};


   // Update the first value
   // of the array
   arr[0] = 1;
   printf("%d", arr[0]);
   return 0;
}
```

Output

1

C Array Traversal

Array Traversal is the process in which we visit every element of the array in a specific order. For C array traversal, we use **loops** to iterate through each element of the array.



Traversing An Array

Example:

```
#include <stdio.h>


int main() {
   int arr[5] = {2, 4, 8, 12, 16};


   // Print each element of
   // array using loop
   printf("Printing Array Elements\n");
   for(int i = 0; i < 5; i++){
      printf("%d ", arr[i]);
   }
   printf("\n");
   // Printing array element in reverse
   printf("Printing Array Elements in Reverse\n");
   for(int i = 4; i>=0; i--){
      printf("%d ", arr[i]);
   }
```

```
    return 0;
}
```

Output

2 4 8 12 16

Size of Array

The size of the array refers to the number of elements that can be stored in the array. The array does not contain the information about its size but we can extract the size using sizeof() operator.

Example:

```
#include <stdio.h>

int main() {
    int arr[5] = {2, 4, 8, 12, 16};

    // Size of the array
    int size = sizeof(arr)/sizeof(arr[0]);
    printf("%d", size);
    return 0;
}
```

The sizeof() operator returns the size in bytes. sizeof(arr) returns the total number of bytes of the array. In an array, each element is of type int, which is 4 bytes. Therefore, we can calculate the size of the array by dividing the total number of bytes by the byte size of one element.

Note: *This method only works in the scope in which the array is declared. Refer to this article to know more -* *Length of Array in C*

Arrays and Pointers

Arrays and Pointers are closely related to each other such that we can use pointers to perform all the possible operations of the array. The array name is a constant pointer to the first element of the array and the array decays to the pointers when passed to the function.

```
#include <stdio.h>

int main() {

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

```
    int* ptr = &arr[0];

    // Address store inside
    // name
    printf("%p\n", arr);

    // Print the address which
    // is pointed by pointer ptr
    printf("%p\n", ptr);
    return 0;
}
```

Output

0x7ffde73e54b0

0x7ffde73e54b0

To know more about the relationship between an array and a pointer, refer to this article - Pointer to an Arrays | Array Pointer.

Passing Array to Function

In C, arrays are passed to functions using pointers, as the array name decays to a pointer to the first element. So, we also need to pass the size of the array to the function.

Example:

```
#include <stdio.h>

// Functions that take array as argument
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
}

int main() {
    int arr[] = {2, 4, 8, 12, 16};
    int n = sizeof(arr) / sizeof(arr[0]);
```

```
    // Passing array
    printArray(arr, n);
    return 0;
}
```

Output

2 4 8 12 16

Multidimensional Array in C

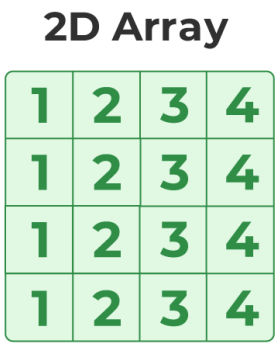
Multi-dimensional arrays in C are arrays that grow in multiple directions or dimensions. A one-dimensional array grows linearly, like parallel to the X-axis, while in a multi-dimensional array, such as a two-dimensional array, the elements can grow along both the X and Y axes.

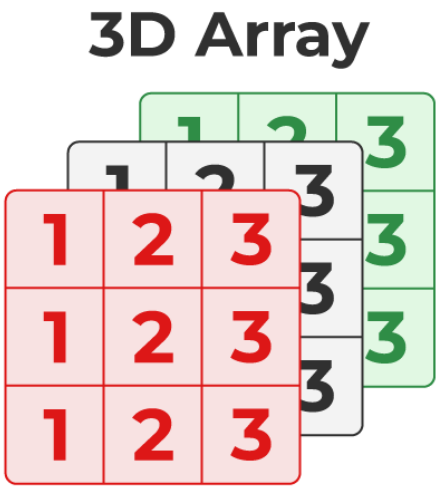
Syntax

type array_name[size1][size2] .. [sizen];

Some commonly used multidimensional arrays are:

- **Two-Dimensional Array**: It is an array that has exactly two dimensions. It can be visualized in the form of rows and columns organized in a two-dimensional plane.
- **Three-Dimensional Array**: A 3D array has exactly three dimensions. It can be visualized as a collection of 2D arrays stacked on top of each other to create the third dimension.





Properties of C Arrays

C arrays have the following distinguishing properties:

1. Fixed Size Collection
2. Homogeneous Elements
3. Indexing in Array
4. Dimensions of Array
5. Contiguous Storage
6. Random Access
7. Array name relation with pointer
8. Bound Checking
9. Array Decay

Advantages of Array in C

The following are the main advantages of an array:

1. Random and fast access of elements using the array index.
2. Use of fewer lines of code as it creates a single array of multiple elements.
3. Traversal through the array becomes easy using a single loop.
4. Sorting becomes easy as it can be accomplished by writing fewer lines of code.

Disadvantages of Array in C

1. C Arrays are not dynamic they only allow a fixed number of elements to be entered which is decided at the time of declaration.
2. Insertion and deletion of elements can be costly since the elements are needed to be rearranged after insertion and deletion.

Properties of Array in C

An array in C is a fixed-size homogeneous collection of elements stored at a contiguous memory location. It is a derived data type in C that can store elements of different data types such as int, char, struct, etc. It is one of the most popular data types widely used by programmers to solve different problems not only in C but also in other languages.

The properties of the arrays depend on the programming language. In this article, we will study the different properties of Array in the C programming language.

1. Fixed Size Collection
2. Homogeneous Elements

3. Indexing in Array
4. Dimensions of Array
5. Contiguous Storage
6. Random Access
7. Array name relation with pointer
8. Bound Checking
9. Array Decay

C Array Properties

1. Fixed Size of an Array

In C, the size of an array is fixed after its declaration. It should be known at the compile time and it cannot be modified later in the program. The below example demonstrates the fixed-size property of the array.

Example:

```
// C Program to Illustrate the Fixed Size Properties of the
// Array
#include <stdio.h>

int main()
{

  // creating a new array of size 5
  int array[5] = { 1, 2, 3, 4, 5 };

  printf("Size of Array Before: %d\n",
      sizeof(array) / sizeof(int));

  // trying to increase the size of the array
  array[6];
  // not checking the size
  printf("Size of Array After: %d",
      sizeof(array) / sizeof(int));
```

```
    return 0;
}
```

Output

Size of Array Before: 5

Size of Array After: 5

2. Homogeneous Collection

An array in C cannot have elements of different data types. All the elements are of the same type.

Example:

```
// C program to Demonstrate the Homogeneous Property of the
// C Array
#include <stdio.h>

int main()
{

    // declaring integer array
    int arr[3] = { 1, 2 };

    // trying to store string in the third element
    arr[2] = "Geeks";

  // printing elements
    printf("Array[1]: %d\n", arr[0]);
    printf("Array[2]: %d\n", arr[1]);
    printf("Array[3]: %s", arr[2]);

    return 0;
}
```

Output

main.c: In function ‘main’:

main.c:12:16: warning: assignment to ‘int’ from ‘char *’ makes integer from pointer without a cast [-Wint-conversion]

```
 12 |       arr[2] = "Geeks";
    |                ^
```

main.c:17:28: warning: format ‘%s’ expects argument of type ‘char *’, but argument 2 has type ‘int’ [-Wformat=]

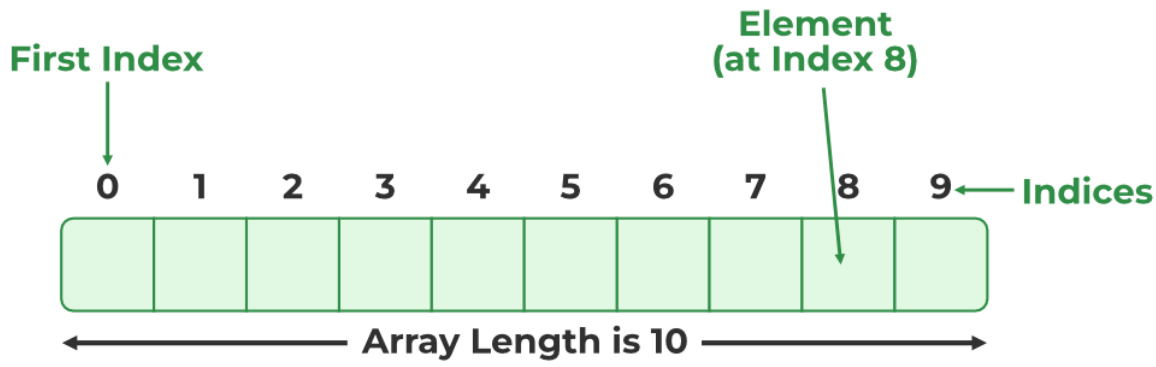
```
 17 |       printf("Array[3]: %s", arr[2]);
    |                          ~^   ~~~~~~
    |                           |    |
    |                           char * int
    |                          %d
```

Array[1]: 1

Array[2]: 2

3. Indexing in an Array

Indexing of elements in an Array in C starts with 0 instead of 1. It means that the index of the first element will be 0 and the last element will be (size - 1) where size is the size of the array.



Indexing of Elements in C Array

Example:

// C Program to Illustrate Array Indexing in C

#include <stdio.h>

```
int main()
{

   // creating integer array with 2 elements
   int arr[2] = { 10, 20 };

   // printing element at index 1
   printf("Array[1]: %d\n", arr[1]);

   // printing element at index 0
   printf("Array[0]: %d", arr[0]);

   return 0;
}
```

Output

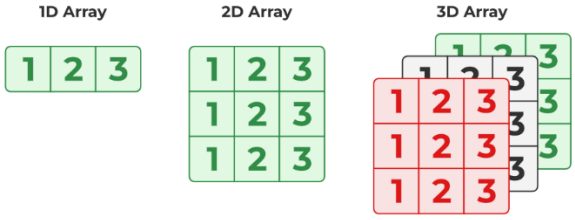
Array[1]: 20

Array[0]: 10

As we see in the above example, at index 1, the second element is present while at index 0, the first element is present.

4. Dimensions of the Array

An array in C can be a single dimensional like a 1-D array or multidimensional like a 2-D array, 3-D array, and so on. It can have any number of dimensions. The number of elements in a multidimensional array is the product of the size of all the dimensions.



Arrays of Different Dimensions

Example:

```c
// C Program to create multidimensional array
#include <stdio.h>

int main()
{

    // creating 2d array
    int arr2d[2][2] = { 1, 2, 3, 4 };

    // creating 3d array
    int arr3d[2][2][2] = { 1, 2, 3, 4, 5, 6, 7, 8 };

    printf("2D Array: ");
    // printing 2d array
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            printf("%d ", arr2d[i][j]);
        }
    }

    printf("\n3D Array: ");
    // printing 3d array
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            for (int k = 0; k < 2; k++) {
                printf("%d ", arr3d[i][j][k]);
            }
        }
    }

    return 0;
```

```
}
```

Output

2D Array: 1 2 3 4

3D Array: 1 2 3 4 5 6 7 8

5. Contiguous Storage

All the elements in an array are stored at contiguous or consecutive memory locations. We can easily imagine this concept in the case of a 1-D array but multidimensional arrays are also stored contiguously. It is possible by storing them in row-major or column-major order where the row after row or column after the column is stored in the memory. We can verify this property by using pointers.

Example:

```
// C Program to Verify the Contiguous Storage of Elements in

// an Array

#include <stdio.h>


int main()

{


   // creating an array of 5 elements

   int arr[5] = { 1, 2, 3, 4, 5 };


   // defining pointers to 2 consecutive elements

   int* ptr1 = &arr[1];

   int* ptr2 = &arr[2];


   // printing the address of arr[1] and arr[2]

   printf("Address of arr[1] : %p\n", ptr1);

   printf("Address of arr[2] : %p", ptr2);


   return 0;

}
```

Output

Address of arr[1] : 0x7fffb8cc1ef4

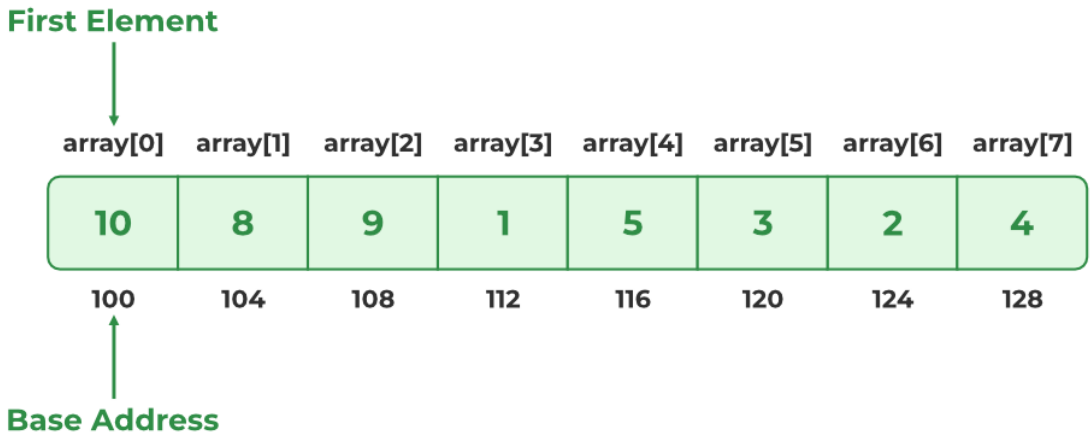
Address of arr[2] : 0x7fffb8cc1ef8

In the above example, the difference between the addresses of arr[1] and arr[2] is 4 bytes which is the memory required to store a single integer. So, at memory addresses 0x7ffebc02e054 to 0x7ffebc02e057, arr[1] is stored and in the next 4 bytes, arr[2] is stored. The same is true for all the elements.

6. Random Access to the Elements

It is one of the defining properties of an Array in C. It means that we can randomly access any element in the array without touching any other element using its index. This property is the result of Contiguous Storage as a compiler deduces the address of the element at the given index by using the address of the first element and the index number.

Address of ith = Address of 1st Element + (Index * Size of Each Element)



Array in C

We can verify this by using Pointer Arithmetic.

Example:

```
// C Program to check the random access property of the
// array
#include <stdio.h>


int main()
{
```

```
    // creating an array of 5 elements
    int arr[5] = { 1, 2, 3, 4, 5 };

    // address of first element
    int* ptr = &arr[0];

    // printing arr[3]
    printf("Array[3]: %d\n", arr[3]);

    // printing element at index 3 using ptr
    printf("Array[3] using pointer to first element = %d",
        *(ptr + 3));

    return 0;
}
```

Output

Array[3]: 4

Array[3] using pointer to first element = 4

Note: We have not multiplied the size of each element in the code as the compiler deduce and multiply it automatically by itself whenever we perform pointer arithmetic.

7. Relationship between Array and Pointers

Arrays are closely related to pointers in the sense that we can do almost all the operations possible on an array using pointers. The array's name itself is the pointer to its first element.

Example:

```
// C Program to Illustrate the Relationship Between Array
// and Pointers
#include <stdio.h>

int main()
{
```

```
    // creating an array with 3 elements
    int arr[3] = { 1, 2, 3 };

    int* ptr = &arr[0];

    // Pointer to first element
    printf("Pointer to First Element: %p\n", ptr);

    // Array name as pointer
    printf("Arran Name: %p", arr);

    return 0;
}
```

Output

Pointer to First Element: 0x7ffec5059660

Arran Name: 0x7ffec5059660

The relationship between pointer and array is very deep and we can study more about it in other articles such as - Pointer to an Array | Array Pointer

8. Bound Checking

Bound checking is the process in which it is checked whether the referenced element is present within the declared range of the Array. In C language, array bound checking is not performed so we can refer to the elements outside the declared range of the array leading to unexpected errors.

Example:

```
// C Program to Illustrate the Out of Bound access in arrays
#include <stdio.h>

int main()
{

    // creating new array with 3 elements
```

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

    // trying to access out of bound element
    printf("Some Garbage Value: %d", arr[5]);

    return 0;
}
```

Output

Some Garbage Value: 0

As seen in the above example, there is no error shown by the compiler while accessing memory that is out of array bounds.

9. Array Decay

Array decay is the process in which an array in C loses its dimension in certain conditions and decays into pointers. After this, we cannot determine the size of the array using sizeof() operator. It happens when an array is passed as a pointer.

Example:

```
// C Program to Demonstrate the Array Decay
#include <stdio.h>

// function
void func(int* arr)
{
    printf("Sizeof Value in Function: %d", sizeof(arr));
}

int main()
{

    // creating array with 3 elements
    char arr[3];
```

```
    printf("Sizeof Value in Main: %d\n", sizeof(arr));

    // passing array
    func(arr);

    return 0;
}
```

Output

Sizeof Value in Main: 3

Sizeof Value in Function: 8

The size of the array in the main() is 3 bytes which is the actual size of the array but when we check the size of the array in func(), the size comes out to be 8 bytes which instead of being the size of the array, it is the size of the pointer to the first element of the array.

Length of Array in C

The Length of an array in C refers to the maximum number of elements that an array can hold. It must be specified at the time of declaration. It is also known as the size of an array that is used to determine the memory required to store all of its elements.

In C language, we don't have any pre-defined function to find the length of the array instead, you must calculate the length manually using techniques based on how the array is declared and used.

The simplest method to find the length of an array is by using **sizeof() operator** to get the total size of the array and then divide it by the size of one element. Take a look at the below example:

```
#include <stdio.h>

int main() {
    int arr[5] = { 1, 2, 3, 4, 5 };

    // Find the size of the array arr
    int n = sizeof(arr) / sizeof(arr[0]);

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

Output

5

Explanation: In this program, the total size of the array **arr** (20 bytes) is divided by the size of a single element in the array (4 bytes). This gives the number of elements in the array, which is **20/4 = 5**

Using Pointer Arithmetic Trick

We can also calculate the length of an array in C using **pointer arithmetic**. This solution of using a pointer is just a hack that is used to find the number of elements in an array.

```
#include <stdio.h>

int main(){


  int arr[5] = { 1, 2, 3, 4, 5 };


  // Find the size of array arr
  int n = *(&arr + 1) - arr;


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

Output

5

Explanation: In the above code, in the expression: ***(&arr + 1) - arr;** is the main logic. Here is how it works,

- **&arr** gives the address of the entire array.
- **&arr + 1** moves the pointer to the address just after the end of the array arr.
- ***(&arr + 1)** dereferences this pointer to get the memory location just after the array.
- Subtracting the base address of the array (**arr**) from this value gives the total number of elements in the array, as pointer arithmetic works based on the size of the elements (e.g., int in this case).

***Note:** Please note that these methods only works when the array is declared in the same scope. These methods will fail if we try them on an array which is passed as a pointer. This happens due to Array Decay. So, it is recommended to keep a variable that tracks the size of the array.*

Multidimensional Arrays in C - 2D and 3D Arrays

A **multi-dimensional array in C** can be defined as an array that has more than one dimension. Having more than one dimension means that it can grow in multiple directions. Some popular multidimensional arrays include 2D arrays which grows in two dimensions, and 3D arrays which grows in three dimensions.

Syntax

The general form of declaring **N-dimensional arrays** is shown below:

type arrName[size1][size2]....[sizeN];

- **type**: Type of data to be stored in the array.
- **arrName**: Name assigned to the array.
- **size1, size2,..., sizeN**: Size of each dimension.

Examples:

// Two-dimensional array

int two_d[10][20];

// Three-dimensional array:

int three_d[10][20][30];

Size of Multidimensional Arrays

The total number of elements that can be stored in a multidimensional array can be calculated by multiplying the size of both dimensions. Consider the array **arr[10][20]:**

- The array **int arr[10][20]** can store total of (10*20) = 200 elements.

To get the size in bytes, we multiply the size of a single element (in bytes) by the total number of elements in the array.

- The size of array **int arr[10][20] = 10 * 20 * 4 = 800 bytes,** where the size of int is 4 bytes.

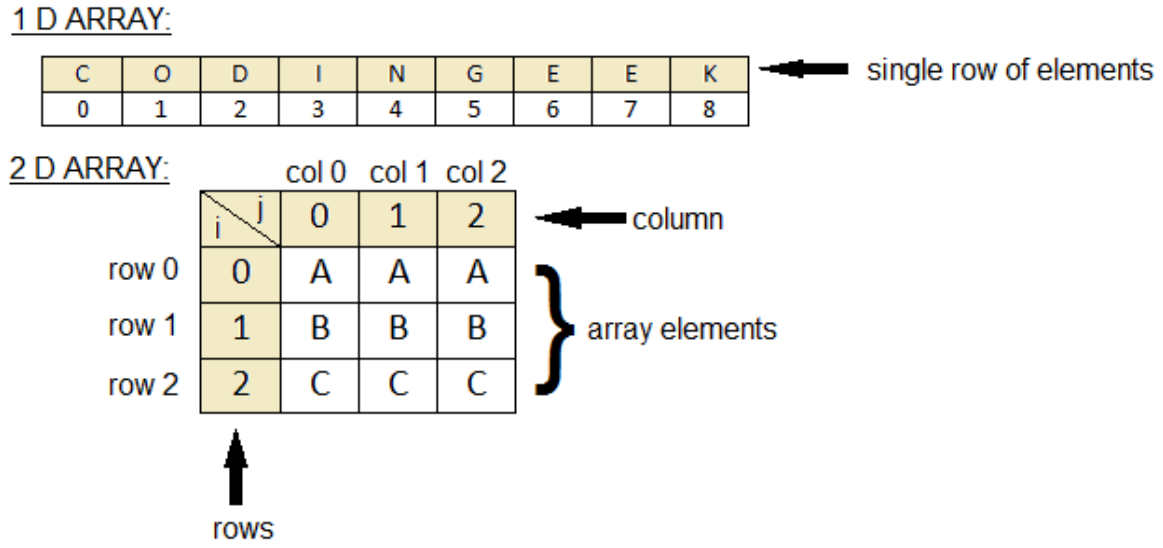
Types of Multidimensional Arrays

In C, there can be many types of arrays depending on their dimensions but two of them are most commonly used:

1. 2D Array - Two Dimensional
2. 3D Array - Three Dimensional

2D Arrays in C

A **two-dimensional array** or **2D array** is the simplest form of the multidimensional array. We can visualize a two-dimensional array as one-dimensional arrays stacked vertically forming a table with 'm' rows and 'n' columns. In C, arrays are 0-indexed, so the row number ranges from 0 to (m-1) and the column number ranges from 0 to (n-1).



Declaration of 2D Array

A 2D array with **m** rows and **n** columns can be created as:

type arr_name[m][n];

For example, we can declare a two-dimensional integer array with name **'arr'** with 10 rows and 20 columns as:

int arr[10][20];

Initialization of 2D Arrays

We can **initialize a 2D array** by using a list of values enclosed inside '{ }' and separated by a comma as shown in the example below:

int arr[3][4] = {0, 1 ,2 ,3 ,4 , 5 , 6 , 7 , 8 , 9 , 10 , 11};

or

int arr[3][4] = {{0, 1, 2, 3}, {4, 5, 6, 7}, {8, 9, 10, 11}};

The elements will be stored in the array from left to right and top to bottom. So, the first 4 elements from the left will be filled in the first row, the next 4 elements in the second row, and so on. This is clearly shown in the second syntax where each set of inner braces represents one row.

In list initialization, we can skip specifying the size of the row. The compiler will automatically deduce it in this case. So, the below declaration is valid.

type arr_name[][n] = {...values...};

It is still compulsory to define the number of columns.

Note: *The number of elements in initializer list should always be either less than or equal to the total number of elements in the array.*

Accessing Elements

An element in two-dimensional array is accessed using row indexes and column indexes inside array subscript **operator []**.

arr_name[i][j]

2D Array Traversal

Traversal means accessing all the elements of the array one by one. We will use two loops, outer loop to go over each row from top to bottom and the inner loop is used to access each element in the current row from left to right.

```
for(int i = 0; i < row; i++){
   for(int j = 0; J < col; j++){
      arr[i][j];
   }
}
```

The below example demonstrates the row-by-row traversal of a 2D array.

```
#include <stdio.h>

int main() {

   // Create and initialize an array with 3 rows
   // and 2 columns
   int arr[3][2] = { { 0, 1 }, { 2, 3 }, { 4, 5 } };

   // Print each array element's value
   for (int i = 0; i < 3; i++) {
      for (int j = 0; j < 2; j++) {
         printf("arr[%d][%d]: %d    ", i, j, arr[i][j]);
      }
      printf("\n");
   }
   return 0;
}
```

Output

arr[0][0]: 0 arr[0][1]: 1

arr[1][0]: 2 arr[1][1]: 3

arr[2][0]: 4 arr[2][1]: 5

How 2D Arrays are Stored in the Memory?

As an array, the elements of the 2D array have to be stored contiguously in memory. As the computers have linear memory addresses, the 2-D arrays must be linearized so as to enable their storage. There are two ways to achieve this:

- **Row Major Order:** This technique stores the 2D array row after row. It means that the first row is stored first in the memory, then the second row of the array, then the third row, and so on.
- **Column Major Column:** This technique stores the 2D array column after column. It means that first column is stored first in the memory, then the second column, then the third column, and so on.

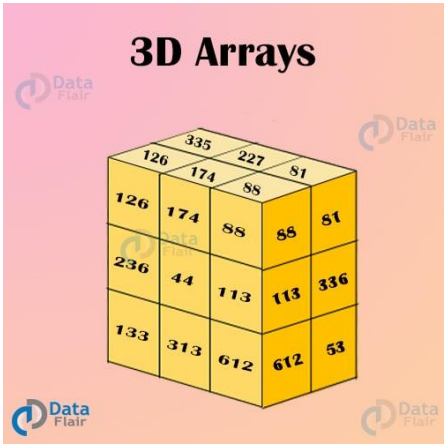
This allows the random access to the 2D array elements as the computer need not keep track of the addresses of all the elements of the array. It only tracks of the Base Address (starting address of the very first element) and calculates the addresses of other array elements using the base address.

Passing 2D Arrays to Functions

Passing 2D arrays to functions need a specialized syntax so that the function knows that the data being passed is 2d array. The function signature that takes 2D array as argument is shown below:

3D Array in C

A **Three-Dimensional Array** or **3D array in C** is a collection of two-dimensional arrays. It can be visualized as multiple 2D arrays stacked on top of each other.



Declaration of 3D Array in C

We can declare a 3D array with **x** 2D arrays each having **m** rows and **n** columns using the syntax shown below:

type arr_name[x][m][n];

For example, we can declare 3d array, which is made by 2-2D array and each 2D array have 2 rows and 2 columns:

int arr[2][2][2];

Initialization of 3D Array in C

Initialization in a 3D array is the same as that of 2D arrays. The difference is as the number of dimensions increases so the number of nested braces will also increase.

int arr[2][3][2] = {0, 1, 2, 3, 4, 5, 6, 7 , 8, 9, 10, 11}

or

int arr[2][3][2] = { { { 1, 1 }, { 2, 3 }, { 4, 5 } },

{ { 6, 7 }, { 8, 9 }, { 10, 11 } } };

Again, just like the 2D arrays, we can also declare the 3D arrays without specifying the size of the first dimensions if we are using initializer list. The compiler will automatically deduce the size of the first dimension. But **we still need to specify the rest of the dimensions.**

arr [][2][2] = {...Values...};

Accessing Elements

To access elements in 3D array, we use three indexes. One for depth, one for row and one for column inside subscript operator [].

arr_name[d][i][j]

where, d, i and j are the indexes for depth (representing a specific 2D array.), the row within that 2D array, and the column within that 2D array respectively.

3D Array Traversal

To traverse the entire 3D array, you need to use three nested loops: an outer loop that goes through the depth (or the set of 2D arrays), a middle loop goes through the rows of each 2D array and at last an inner loop goes through each element of the current row.

```
for(int d = 0; d < depth; d++){
  for(int i = 0; i < row; i++){
    for(int j = 0; j < col; j++){
      arr[d][i][j];
    }
  }
}
```

Let's take a simple example to demonstrate all the concepts we discussed about 3D arrays:

#include <stdio.h>

```
int main() {

  // Create and Initialize the
  // 3-dimensional array
  int arr[2][3][2] = { { { 1, 1 }, { 2, 3 },
                { 4, 5 } }, { { 6, 7 },
                { 8, 9 }, { 10, 11 } } };

  // Loop through the depth
  for (int i = 0; i < 2; ++i) {

    // Loop through the
    // rows of each depth
    for (int j = 0; j < 3; ++j) {

      // Loop through the
      // columns of each row
      for (int k = 0; k < 2; ++k)
        printf("arr[%i][%i][%i] = %d   ", i, j, k,
            arr[i][j][k]);
      printf("\n");
    }
   printf("\n\n");
  }
  return 0;
}
```

Output

```
arr[0][0][0] = 1   arr[0][0][1] = 1
arr[0][1][0] = 2   arr[0][1][1] = 3
arr[0][2][0] = 4   arr[0][2][1] = 5
```

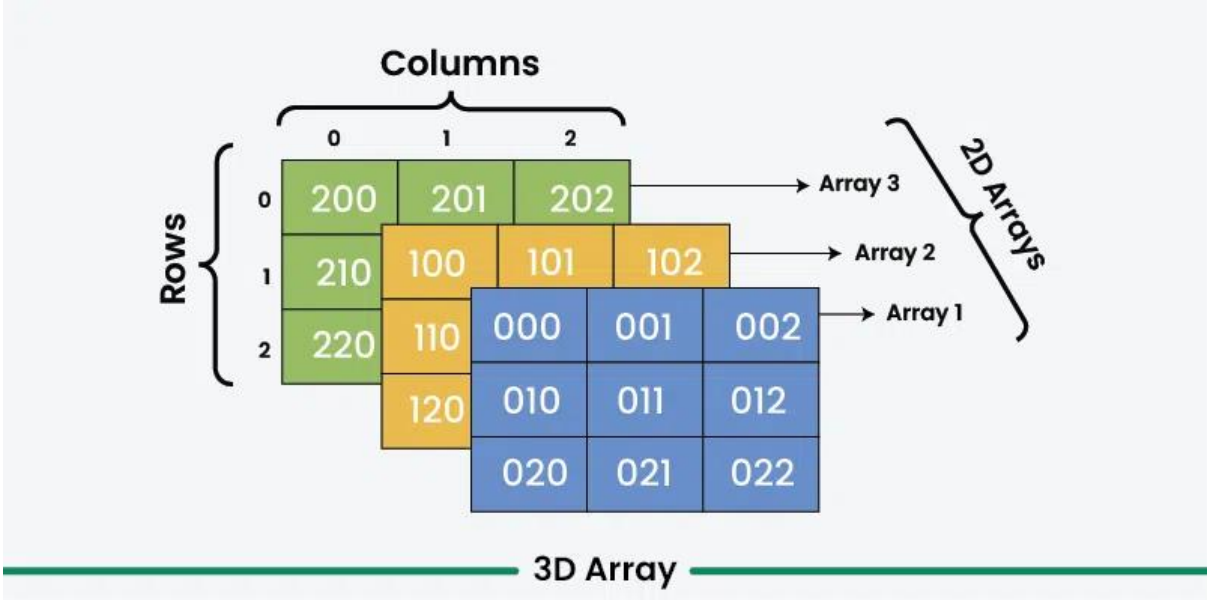
arr[1][0][0] = 6 arr[1][0][1] = 7

arr[1][1][0] = 8 arr[1][1][1] = 9

arr[1][2][0] = 10 arr[1][2][1] = 11

How 3D Arrays are Stored in the Memory?

Like 2D arrays, the elements of a 3D array should also be stored contiguously in memory.



3D Array

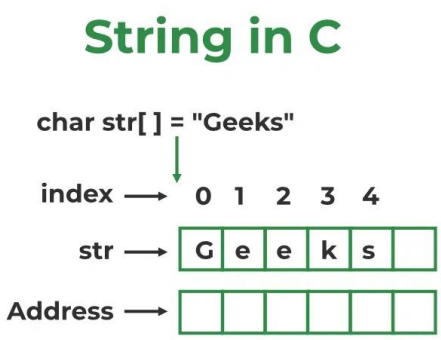
Since computers have linear memory, the 3D array must also be linearized for storage. We use the same two techniques, the **Row Major Order and Column Major Order** but with added dimension. The elements are first stored layer by layer (or 2D array by 2D array). Within each 2D array, the elements follow the corresponding row or column major order.

Passing 3D Arrays to Functions

Passing a 3D array to a function in C is similar to passing 2D arrays, but with an additional dimension. When passing a 3D array, you need to pass the sizes of all the dimensions separately because the size information of array is lost while passing.

Strings in C

A String in C programming is a sequence of characters terminated with a null character '\0'. The C String is work as an array of characters. The difference between a character array and a C string is that the string in C is terminated with a unique character **'\0'**.



Declaration

Declaring a string in C is as simple as declaring a one-dimensional array of character type. Below is the syntax for declaring a string.

char string_name[size];

In the above syntax **string_name** is any name given to the string variable and size is used to define the length of the string, i.e. the number of characters strings will store.

Like array, we can skip the size in the above statement:

char array_name[];

Initialization

We can initialize a string either by specifying the list of characters or string literal.

// Using character list

char str[] = {'G', 'e', 'e', 'k', 's', '\0'};

// Using string literal

char str[] = "Geeks";

Note: *When a Sequence of characters enclosed in the double quotation marks is encountered by the compiler, a null character '\0' is appended at the end of the string by default.*

Accessing

We can access any character of the string by providing the position of the character, like in array. We pass the index inside square brackets [] with the name of the string.

Example:

```
#include <stdio.h>


int main() {


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

```
    // Access first character
    // of string
    printf("%c", str[0]);
    return 0;
}
```

Update

Updating a character in a string is also possible. We can update any character of a string by using the index of that character.

Example:

```
#include <stdio.h>

int main() {
    char str[] = "Geeks";

    // Update the first
    // character of string
    str[0] = 'R';
    printf("%c", str[0]);
    return 0;
}
```

Output

R

String Length

To find the length of a string, you need to iterate through it until you reach the null character ('\0'). The strlen() function from the C standard library is commonly used to get the length of a string.

Example:

```
#include <stdio.h>

int main() {
```

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

    printf("%d", strlen(str));
    return 0;
}
```

Output

5

In this example, **strlen()** returns the length of the string "**Geeks**", which is **5**, excluding the null character.

String Input

In C, reading a string from the user can be done using different functions, and depending on the use case, one method might be chosen over another. Below, the common methods of reading strings in C will be discussed, including how to handle whitespace, and concepts will be clarified to better understand how string input works.

Using scanf()

The simplest way to read a string in C is by using the scanf() function.

Example:

```
#include<stdio.h>

int main() {
    char str[5];

    // Read string
    // from the user
    scanf("%s",str);

    // Print the string
    printf("%s",str);
    return 0;
}
```

Output

Geeks *(Enter by user)*
Geeks

In the above program, the string is taken as input using the scanf() function and is also printed. However, there is a limitation with the scanf() function. scanf() will stop reading input as soon as it encounters a whitespace (space, tab, or newline).

Using scanf() with a Scanset

We can also use **scanf()** to read strings with spaces by utilizing a **scanset**. A scanset in scanf() allows specifying the characters to include or exclude from the input.

Example:

```
#include <stdio.h>


int main() {
    char str[20];


    // Using scanset in scanf
    // to read until newline
    scanf("%[^\n]s", str);


    // Printing the read string
    printf("%s", str);


    return 0;
}
```

Output

Geeks For Geeks *(Enter by user)*
Geeks For Geeks

Using fgets()

If someone wants to read a complete string, including spaces, they should use the **fgets()** function. Unlike scanf(), fgets() reads the entire line, including spaces, until it encounters a newline.

Example:

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

```
int main() {
    char str[20];

    // Reading the string
    // (with spaces) using fgets
    fgets(str, 20, stdin);

    // Displaying the string using puts
    printf("%s", str);
    return 0;
}
```

Output

Geeks For Geeks *(Enter by user)*
Geeks For Geeks

Passing Strings to Function

As strings are character arrays, we can pass strings to functions in the same way we **pass an array to a function**. Below is a sample program to do this:

```
#include <stdio.h>

void printStr(char str[]) {
    printf("%s", str);
}

int main() {
    char str[] = "GeeksforGeeks";

    // Passing string to a
    // function
    printStr(str);
    return 0;
}
```

Output

GeeksforGeeks

Strings and Pointers in C

Similar to arrays, In C, we can create a character pointer to a string that points to the starting address of the string which is the first character of the string. The string can be accessed with the help of pointers as shown in the below example.

```
#include <stdio.h>


int main(){


  char str[20] = "Geeks";


  // Pointer variable which stores
  // the starting address of
  // the character array str
  char* ptr = str;


  // While loop will run till
  // the character value is not
  // equal to null character
  while (*ptr != '\0') {
    printf("%c", *ptr);
    ptr++;
  }
  return 0;
}
```

Output

Geeks

Standard C Library - String.h Functions

The C language comes bundled with **<string.h>** which contains some useful string-handling functions. Some of them are as follows:

Function	Description
strlen()	Returns the length of string name.
strcpy()	Copies the contents of string s2 to string s1.
strcmp()	Compares the first string with the second string. If strings are the same it returns 0.
strcat()	Concat s1 string with s2 string and the result is stored in the first string.
strlwr()	Converts string to lowercase.
strupr()	Converts string to uppercase.
strstr()	Find the first occurrence of s2 in s1.

UNIT IV

Pointers & User Defined Data types

Pointers, dereferencing and address operators, pointer and address arithmetic, array manipulation using pointers, User-defined data types-Structures and Unions.

C Pointers

A **pointer** is a variable that stores the **memory address** of another variable. Instead of holding a direct value, it has the address where the value is stored in memory. This allows us to manipulate the data stored at a specific memory location without actually using its variable. It is the backbone of low-level memory manipulation in C.

Declare a Pointer

A pointer is declared by specifying its name and type, just like simple variable declaration but with an **asterisk (*)** symbol added before the pointer's name.

data_type* name

Here, **data_type** defines the type of data that the pointer is pointing to. An integer type pointer can only point to an integer. Similarly, a pointer of float type can point to a floating-point data, and so on.

Example:

int *ptr;

In the above statement, pointer **ptr** can store the address of an integer. It is pronounced as pointer to integer.

Initialize the Pointer

Pointer initialization means assigning some address to the pointer variable. In C, the **(&) addressof operator** is used to get the memory address of any variable. This memory address is then stored in a pointer variable.

Example:

int var = 10;

// Initializing ptr

int *ptr = &var;

In the above statement, pointer **ptr** store the address of variable **var** which was determined using address-of operator **(&).**

Note: *We can also declare and initialize the pointer in a single step. This is called* ***pointer definition.***

Dereference a Pointer

Accessing the pointer directly will just give us the address that is stored in the pointer. For example,

```
#include <stdio.h>


int main() {
  int var = 10;


  // Store address of var variable
  int* ptr = &var;


  // Directly accessing ptr
  printf("%d", ptr);


  return 0;
}
```

Output

0x7fffa0757dd4

This hexadecimal integer (starting with 0x) is the memory address.

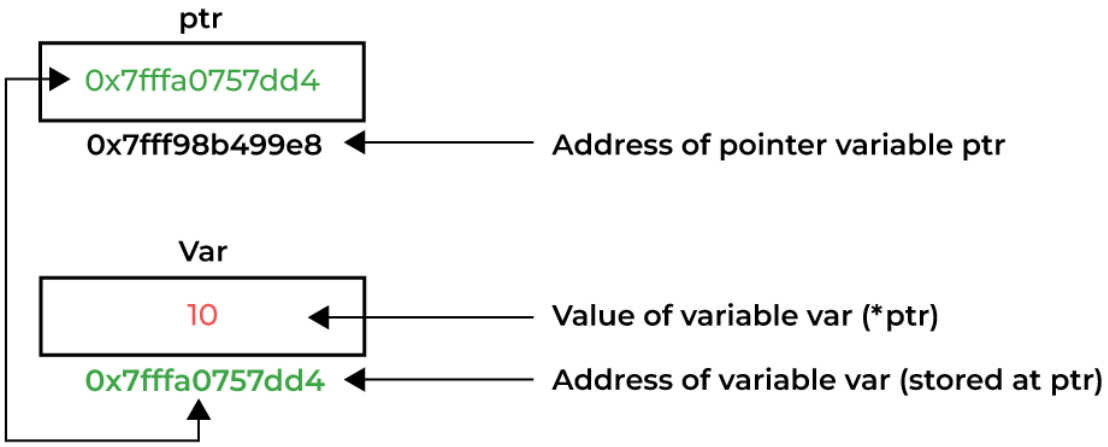
We have to first **dereference** the pointer to access the value present at the memory address. This is done with the help of **dereferencing operator(*)** (same operator used in declaration).

```
#include <stdio.h>


int main() {
  int var = 10;


  // Store address of var variable
  int* ptr = &var;


  // Dereferencing ptr to access the value
  printf("%d", *ptr);
```

```
    return 0;
}
```

Output

```
10
```



***Note**: Earlier, we used %d for printing pointers, but C provides a separate format specifier **%p** for printing pointers.*

Size of Pointers

The **size of a pointer in C** depends on the **architecture (bit system)** of the machine, **not the data type** it points to.

- On a **32-bit system**, all pointers typically occupy **4 bytes**.
- On a **64-bit system**, all pointers typically occupy **8 bytes**.

The size remains **constant regardless of the data type** (int*, char*, float*, etc.). We can verify this using the sizeof operator.

```
#include <stdio.h>


int main() {
    int *ptr1;
    char *ptr2;


    // Finding size using sizeof()
    printf("%zu\n", sizeof(ptr1));
    printf("%zu", sizeof(ptr2));
```

```
    return 0;
}
```

Output

```
8
8
```

The reason for the same size is that the pointers store the memory addresses, no matter what type they are. As the space required to store the addresses of the different memory locations is the same, the memory required by one pointer type will be equal to the memory required by other pointer types.

Note: *The actual size of the pointer may vary depending on the* ***compiler and system architecture****, but it is always* ***uniform across all data types*** *on the same system*.

Special Types of Pointers

There are 4 special types of pointers that used or referred to in different contexts:

NULL Pointer

The NULL Pointers are those pointers that do not point to any memory location. They can be created by assigning **NULL** value to the pointer. A pointer of any type can be assigned the NULL value.

```
#include <stdio.h>

int main() {
    // Null pointer
    int *ptr = NULL;

    return 0;
}
```

NULL pointers are generally used to represent the absence of any address. This allows us to check whether the pointer is pointing to any valid memory location by checking if it is equal to NULL.

Void Pointer

The void pointers in C are the pointers of type **void**. It means that they do not have any associated data type. They are also called **generic pointers** as they can point to any type and can be typecasted to any type.

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

```
int main() {
  // Void pointer
  void *ptr;

  return 0;
}
```

Wild Pointers

The wild pointers are pointers that have not been initialized with something yet. These types of C-pointers can cause problems in our programs and can eventually cause them to crash. If values are updated using wild pointers, they could cause data abort or data corruption.

```
#include <stdio.h>

int main() {

  // Wild Pointer
  int *ptr;

  return 0;
}
```

Dangling Pointer

A pointer pointing to a memory location that has been deleted (or freed) is called a dangling pointer. Such a situation can lead to unexpected behavior in the program and also serve as a source of bugs in C programs.

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

int main() {
  int* ptr = (int*)malloc(sizeof(int));

  // After below free call, ptr becomes a dangling pointer
  free(ptr);
```

```
    printf("Memory freed\n");


    // removing Dangling Pointer
    ptr = NULL;


    return 0;
}
```

Output

Memory freed

C Pointer Arithmetic

The pointer arithmetic refers to the arithmetic operations that can be performed on a pointer. It is slightly different from the ones that we generally use for mathematical calculations as only a limited set of operations can be performed on pointers. These operations include:

- **Increment/Decrement**
- **Addition/Subtraction of Integer**
- **Subtracting Two Pointers of Same Type**
- **Comparing/Assigning Two Pointers of Same Type**
- **Comparing/Assigning with NULL**

C Pointers and Arrays

In C programming language, pointers and arrays are closely related. An array name acts like a pointer constant. The value of this pointer constant is the address of the first element. For example, if we have an array named **val,** then **val** and **&val[0]** can be used interchangeably.

If we assign this value to a non-constant pointer to array of the same type, then we can access the elements of the array using this pointer. Not only that, as the array elements are stored continuously, we can use pointer arithmetic operations such as increment, decrement, addition, and subtraction of integers on pointer to move between array elements.

This concept is not limited to the one-dimensional array, we can refer to a multidimensional array element as well using pointers.

Constant Pointers

In **constant pointers**, the memory address stored inside the pointer is constant and cannot be modified once it is defined. It will always point to the same memory address.

Example:

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

```
int main() {
   int a = 90;
   int b = 50;

   // Creating a constant pointer
   int* const ptr = &a;

   // Trying to reassign it to b
   ptr = &b;

   return 0;
}
```

Output

```
solution.c: In function ‘main’:
solution.c:11:9: error: assignment of read-only variable ‘ptr’
  11 |    ptr = &b;
     |      ^
```

We can also create a pointer to constant or even constant pointer to constant. Refer to this article to know more - Constant pointer, Pointers to Constant and Constant Pointers to Constant

Pointer to Function

A **function pointer** is a type of pointer that stores the address of a function, allowing functions to be passed as arguments and invoked dynamically. It is useful in techniques such as callback functions, event-driven programs.

Example:

```
#include <stdio.h>

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

```
int main() {

    // Declare a function pointer that matches
    // the signature of add() fuction
    int (*fptr)(int, int);

    // Assign address of add()
    fptr = &add;

    // Call the function via ptr
    printf("%d", fptr(10, 5));

    return 0;
}
```

Output

15

Multilevel Pointers

In C, we can create **multi-level pointers** with any number of levels such as – ***ptr3, ****ptr4, ******ptr5 and so on. Most popular of them is **double pointer** (pointer to pointer). It stores the memory address of another pointer. Instead of pointing to a data value, they point to another pointer.

Example:

```
#include <stdio.h>

int main() {
    int var = 10;

    // Pointer to int
    int *ptr1 = &var;

    // Pointer to pointer (double pointer)
```

```
    int **ptr2 = &ptr1;

    // Accessing values using all three
    printf("var: %d\n", var);
    printf("*ptr1: %d\n", *ptr1);
    printf("**ptr2: %d", **ptr2);

    return 0;
}
```

Output

var: 10

*ptr1: 10

**ptr2: 10

Uses of Pointers in C

The C pointer is a very powerful tool that is widely used in C programming to perform various useful operations. It is used to achieve the following functionalities in C:

- Pass Arguments by Pointers
- Accessing Array Elements
- Return Multiple Values from Function
- Dynamic Memory Allocation
- Implementing Data Structures
- In System-Level Programming where memory addresses are useful.
- To use in Control Tables.

Advantages of Pointers

Following are the major advantages of pointers in C:

- Pointers are used for dynamic memory allocation and deallocation.
- An Array or a structure can be accessed efficiently with pointers
- Pointers are useful for accessing memory locations.
- Pointers are used to form complex data structures such as linked lists, graphs, trees, etc.
- Pointers reduce the length of the program and its execution time as well.

Issues with Pointers

Pointers are vulnerable to errors and have following disadvantages:

- Memory corruption can occur if an incorrect value is provided to pointers.
- Pointers are a little bit complex to understand.
- Pointers are majorly responsible for memory leaks in C.
- Accessing using pointers are comparatively slower than variables in C.
- Uninitialized pointers might cause a segmentation fault.

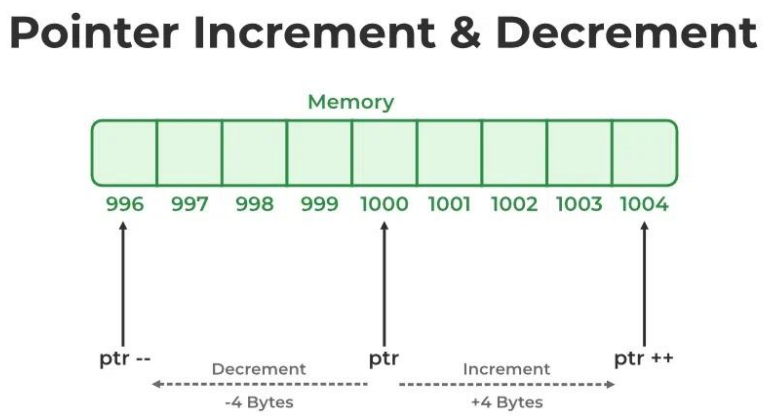
Pointer Arithmetics in C with Examples

Pointer Arithmetic is the set of valid arithmetic operations that can be performed on pointers. The pointer variables store the memory address of another variable. It doesn't store any value.

Hence, there are only a few operations that are allowed to perform on Pointers in C language. The C pointer arithmetic operations are slightly different from the ones that we generally use for mathematical calculations. These operations are:

1. Increment/Decrement of a Pointer
2. Addition of integer to a pointer
3. Subtraction of integer to a pointer
4. Subtracting two pointers of the same type
5. Comparison of pointers

1. **Increment/Decrement of a Pointer**



Increment: It is a condition that also comes under addition. When a pointer is incremented, it actually increments by the number equal to the size of the data type for which it is a pointer.

For Example:
If an integer pointer that stores **address 1000** is incremented, then it will increment by 4(**size of an int**), and the new address will point to **1004**. While if a float type pointer is incremented then it will increment by 4(**size of a float**) and the new address will be **1004**.

Decrement: It is a condition that also comes under subtraction. When a pointer is decremented, it actually decrements by the number equal to the size of the data type for which it is a pointer.

For Example:
If an integer pointer that stores **address 1000** is decremented, then it will decrement by 4(**size of an int**), and the new address will point to **996**. While if a float type pointer is decremented then it will decrement by 4(**size of a float**) and the new address will be **996**.

Note: *It is assumed here that the architecture is 64-bit and all the data types are sized accordingly. For example, integer is of 4 bytes.*

Example of Pointer Increment and Decrement

Below is the program to illustrate pointer increment/decrement:

```
#include <stdio.h>

// pointer increment and decrement

//pointers are incremented and decremented by the size of the data type they point to

int main()

{

  int a = 22;

  int *p = &a;

  printf("p = %u\n", p); // p = 6422288

  p++;

  printf("p++ = %u\n", p); //p++ = 6422292   +4   // 4 bytes

  p--;

  printf("p-- = %u\n", p); //p-- = 6422288    -4   // restored to original value


  float b = 22.22;

  float *q = &b;

  printf("q = %u\n", q);  //q = 6422284

  q++;

  printf("q++ = %u\n", q); //q++ = 6422288     +4   // 4 bytes
```

```
    q--;
    printf("q-- = %u\n", q); //q-- = 6422284      -4  // restored to original value

    char c = 'a';
    char *r = &c;
    printf("r = %u\n", r);   //r = 6422283
    r++;
    printf("r++ = %u\n", r);   //r++ = 6422284    +1   // 1 byte
    r--;
    printf("r-- = %u\n", r);   //r-- = 6422283    -1  // restored to original value

    return 0;
}
```

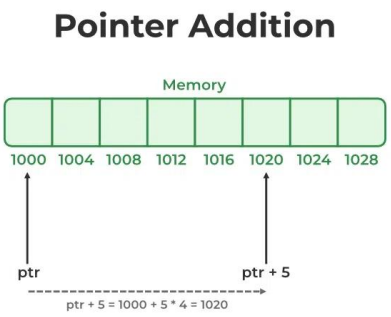
Output

```
p = 1441900792
p++ = 1441900796
p-- = 1441900792
q = 1441900796
q++ = 1441900800
q-- = 1441900796
r = 1441900791
r++ = 1441900792
r-- = 1441900791
```

Note: *Pointers can be outputted using %p, since, most of the computers store the address value in hexadecimal form using %p gives the value in that form. But for simplicity and understanding we can also use %u to get the value in Unsigned int form.*

2. Addition of Integer to Pointer

When a pointer is added with an integer value, the value is first multiplied by the size of the data type and then added to the pointer.



For Example:
Consider the same example as above where the **ptr** is an **integer pointer** that stores **1000** as an address. If we add integer 5 to it using the expression, **ptr = ptr + 5,** then, the final address stored in the ptr will be **ptr = 1000 + sizeof(int) * 5 = 1020.**

Example of Addition of Integer to Pointer

```
// C program to illustrate pointer Addition
#include <stdio.h>

// Driver Code
int main()
{
  // Integer variable
  int N = 4;

  // Pointer to an integer
  int *ptr1, *ptr2;

  // Pointer stores the address of N
  ptr1 = &N;
  ptr2 = &N;

  printf("Pointer ptr2 before Addition: ");
  printf("%p \n", ptr2);

  // Addition of 3 to ptr2
```

```
    ptr2 = ptr2 + 3;
    printf("Pointer ptr2 after Addition: ");
    printf("%p \n", ptr2);

    return 0;
}
```

Output

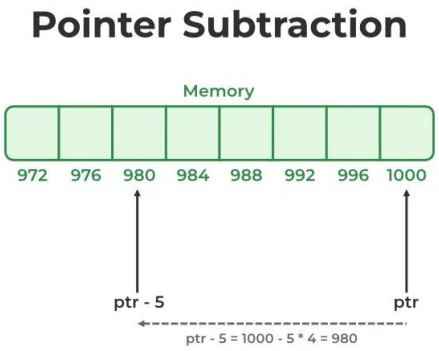
Pointer ptr2 before Addition: 0x7ffca373da9c

Pointer ptr2 after Addition: 0x7ffca373daa8

3. Subtraction of Integer to Pointer

When a pointer is subtracted with an integer value, the value is first multiplied by the size of the data type and then subtracted from the pointer similar to addition.

For Example:
Consider the same example as above where the **ptr** is an **integer pointer** that stores **1000** as an address. If we subtract integer 5 from it using the expression, **ptr = ptr - 5,** then, the final address stored in the ptr will be **ptr = 1000 - sizeof(int) * 5 = 980.**



Example of Subtraction of Integer from Pointer

Below is the program to illustrate pointer Subtraction:

```
// C program to illustrate pointer Subtraction
#include <stdio.h>

// Driver Code
int main()
{
    // Integer variable
```

```
    int N = 4;

    // Pointer to an integer
    int *ptr1, *ptr2;

    // Pointer stores the address of N
    ptr1 = &N;
    ptr2 = &N;

    printf("Pointer ptr2 before Subtraction: ");
    printf("%p \n", ptr2);

    // Subtraction of 3 to ptr2
    ptr2 = ptr2 - 3;
    printf("Pointer ptr2 after Subtraction: ");
    printf("%p \n", ptr2);

    return 0;
}
```

Output

Pointer ptr2 before Subtraction: 0x7ffd718ffebc

Pointer ptr2 after Subtraction: 0x7ffd718ffeb0

4. Subtraction of Two Pointers

The subtraction of two pointers is possible only when they have the same data type. The result is generated by calculating the difference between the addresses of the two pointers and calculating how many bytes of data it is according to the pointer data type. The subtraction of two pointers gives the increments between the two pointers.

For Example:
Two integer pointers say **ptr1(address:1000)** and **ptr2(address:1004)** are subtracted. The difference between addresses is 4 bytes. Since the size of int is 4 bytes, therefore the **increment between ptr1 and ptr2** is given by **(4/4) = 1**.

Example of Subtraction of Two Pointer

Below is the implementation to illustrate the Subtraction of Two Pointers:

```
// C program to illustrate Subtraction
// of two pointers
#include <stdio.h>

// Driver Code
int main()
{
   int x = 6; // Integer variable declaration
   int N = 4;

   // Pointer declaration
   int *ptr1, *ptr2;

   ptr1 = &N; // stores address of N
   ptr2 = &x; // stores address of x

   printf(" ptr1 = %u, ptr2 = %u\n", ptr1, ptr2);
   // %p gives an hexa-decimal value,
   // We convert it into an unsigned int value by using %u

   // Subtraction of ptr2 and ptr1
   x = ptr1 - ptr2;

   // Print x to get the Increment
   // between ptr1 and ptr2
   printf("Subtraction of ptr1 "
        "& ptr2 is %d\n",
        x);

   return 0;
```

```
}
```

Output

```
 ptr1 = 2715594428, ptr2 = 2715594424

Subtraction of ptr1 & ptr2 is 1
```

5. Comparison of Pointers

We can compare the two pointers by using the comparison operators in C. We can implement this by using all operators in C **>, >=, <, <=, ==, !=.** It returns true for the valid condition and returns false for the unsatisfied condition.

1. **Step 1:** Initialize the integer values and point these integer values to the pointer.
2. **Step 2:** Now, check the condition by using comparison or relational operators on pointer variables.
3. **Step 3:** Display the output.

Example of Pointer Comparision

```
// C Program to illustrare pointer comparision

#include <stdio.h>


int main()

{

   // declaring array

   int arr[5];


   // declaring pointer to array name

   int* ptr1 = &arr;

   // declaring pointer to first element

   int* ptr2 = &arr[0];


   if (ptr1 == ptr2) {

      printf("Pointer to Array Name and First Element "

          "are Equal.");

   }

   else {
```

```
    printf("Pointer to Array Name and First Element "
        "are not Equal.");
  }

  return 0;
}
```

Output

Pointer to Array Name and First Element are Equal.

Comparison to NULL

A pointer can be compared or assigned a NULL value irrespective of what is the pointer type. Such pointers are called NULL pointers and are used in various pointer-related error-handling methods.

```
// C Program to demonstrate the pointer comparison with NULL
// value
#include <stdio.h>

int main()
{

  int* ptr = NULL;

  if (ptr == NULL) {
    printf("The pointer is NULL");
  }
  else {
    printf("The pointer is not NULL");
  }
  return 0;
}
```

Output

The pointer is NULL

Comparison operators on Pointers using an array

In the below approach, it results in the count of odd numbers and even numbers in an array. We are going to implement this by using a pointer.

1. **Step 1:** First, declare the length of an array and array elements.
2. **Step 2:** Declare the pointer variable and point it to the first element of an array.
3. **Step 3:** Initialize the count_even and count_odd. Iterate the for loop and check the conditions for the number of odd elements and even elements in an array
4. **Step 4:** Increment the pointer location ptr++ to the next element in an array for further iteration.
5. **Step 5:** Print the result.

Example of Pointer Comparison in Array

```
// Pointer Comparision in Array

#include <stdio.h>


int main()

{

   int n = 10; // length of an array


   int arr[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

   int* ptr; // Declaration of pointer variable


   ptr = arr; // Pointer points the first (0th index)

         // element in an array

   int count_even = 0;

   int count_odd = 0;


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


      if (*ptr % 2 == 0) {

         count_even++;

      }
```

```
        if (*ptr % 2 != 0) {
            count_odd++;
        }
        ptr++; // Pointing to the next element in an array
    }
    printf("No of even elements in an array is : %d",
           count_even);
    printf("\nNo of odd elements in an array is : %d",
           count_odd);
}
```

Output

No of even elements in an array is : 5

No of odd elements in an array is : 5

Pointer Arithmetic on Arrays

Pointers contain addresses. Adding two addresses makes no sense because there is no idea what it would point to. Subtracting two addresses lets you compute the offset between the two addresses. An array name acts like a pointer constant. The value of this pointer constant is the address of the first element.

For Example: if an array is named **arr** then **arr and &arr[0]** can be used to reference the array as a pointer.

Below is the program to illustrate the Pointer Arithmetic on arrays:

Program 1:

```
// C program to illustrate the array
// traversal using pointers
#include <stdio.h>

// Driver Code
int main()
{

    int N = 5;
```

```
    // An array
    int arr[] = { 1, 2, 3, 4, 5 };

    // Declare pointer variable
    int* ptr;

    // Point the pointer to first
    // element in array arr[]
    ptr = arr;

    // Traverse array using ptr
    for (int i = 0; i < N; i++) {

        // Print element at which
        // ptr points
        printf("%d ", ptr[0]);
        ptr++;
    }
}
```

Output

1 2 3 4 5

Program 2:

```
// C program to illustrate the array
// traversal using pointers in 2D array
#include <stdio.h>

// Function to traverse 2D array
// using pointers
void traverseArr(int* arr, int N, int M)
{
```

```
    int i, j;

    // Traverse rows of 2D matrix
    for (i = 0; i < N; i++) {

        // Traverse columns of 2D matrix
        for (j = 0; j < M; j++) {

            // Print the element
            printf("%d ", *((arr + i * M) + j));
        }
        printf("\n");
    }
}

// Driver Code
int main()
{

    int N = 3, M = 2;

    // A 2D array
    int arr[][2] = { { 1, 2 }, { 3, 4 }, { 5, 6 } };

    // Function Call
    traverseArr((int*)arr, N, M);
    return 0;
}
```

Output

1 2

3 4

5 6

Array of Pointers in C

In C, a pointer array is a homogeneous collection of indexed pointer variables that are references to a memory location. It is generally used in C Programming when we want to point at multiple memory locations of a similar data type in our C program. We can access the data by dereferencing the pointer pointing to it.

Syntax:

pointer_type *array_name [array_size];

Here,

- **pointer_type:** Type of data the pointer is pointing to.
- **array_name:** Name of the array of pointers.
- **array_size:** Size of the array of pointers.

Note: *It is important to keep in mind the operator precedence and associativity in the array of pointers declarations of different type as a single change will mean the whole different thing. For example, enclosing *array_name in the parenthesis will mean that array_name is a pointer to an array.*

Example:

```
// C program to demonstrate the use of array of pointers
#include <stdio.h>

int main()
{
  // declaring some temp variables
  int var1 = 10;
  int var2 = 20;
  int var3 = 30;

  // array of pointers to integers
  int* ptr_arr[3] = { &var1, &var2, &var3 };

  // traversing using loop
```

```
    for (int i = 0; i < 3; i++) {
        printf("Value of var%d: %d\tAddress: %p\n", i + 1, *ptr_arr[i], ptr_arr[i]);
    }

    return 0;
}
```

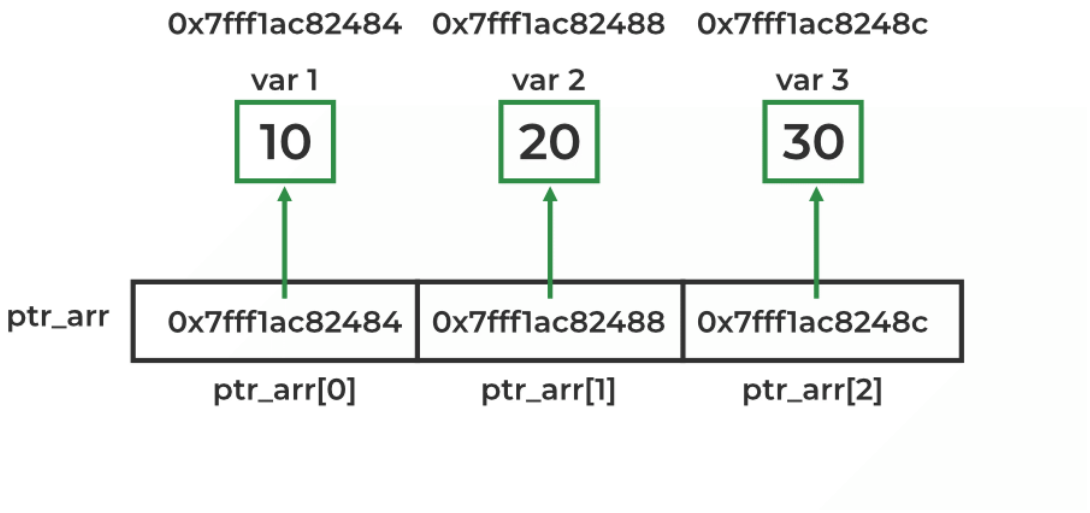
Output

Value of var1: 10 Address: 0x7fff1ac82484

Value of var2: 20 Address: 0x7fff1ac82488

Value of var3: 30 Address: 0x7fff1ac8248c

Explanation:



As shown in the above example, each element of the array is a pointer pointing to an integer. We can access the value of these integers by first selecting the array element and then dereferencing it to get the value.

Array of Pointers to Character

One of the main applications of the array of pointers is to store multiple strings as an array of pointers to characters. Here, each pointer in the array is a character pointer that points to the first character of the string.

Syntax:

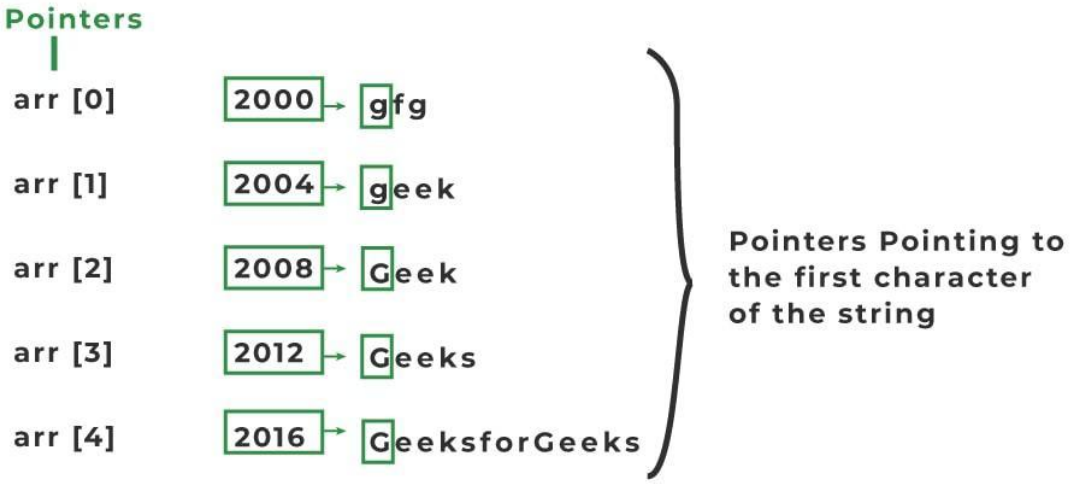
char *array_name [array_size];

After that, we can assign a string of any length to these pointers.

Example:

char* arr[5]

= { "gfg", "geek", "Geek", "Geeks", "GeeksforGeeks" }



Note: *Here, each string will take different amount of space so offset will not be the same and does not follow any particular order.*

This method of storing strings has the advantage of the traditional array of strings. Consider the following two examples:

Example 1:

```
// C Program to print Array of strings without array of pointers

#include <stdio.h>

int main()

{

  char str[3][10] = { "Geek", "Geeks", "Geekfor" };


  printf("String array Elements are:\n");


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

    printf("%s\n", str[i]);

  }


  return 0;
```

```
}
```

Output

```
String array Elements are:
Geek
Geeks
Geekfor
```

In the above program, we have declared the 3 rows and 10 columns of our array of strings. But because of predefining the size of the array of strings the space consumption of the program increases if the memory is not utilized properly or left unused. Now let's try to store the same strings in an array of pointers.

Example 2:

```
// C Program to print Array of strings with array of pointers
#include <stdio.h>

int main() {
  // Declare an array of pointers to characters
  char* arr[] = { "geek", "Geeks", "Geeksfor" };

  // Print each string and its address
  for (int i = 0; i < 3; i++) {
    printf("%s\n", arr[i]);
  }

  // Print addresses of each string
  for (int i = 0; i < 3; i++) {
    printf("Address of arr[%d]: %p\n", i, (void*)arr[i]);
  }

  return 0;
}
```

Output

geek

Geeks

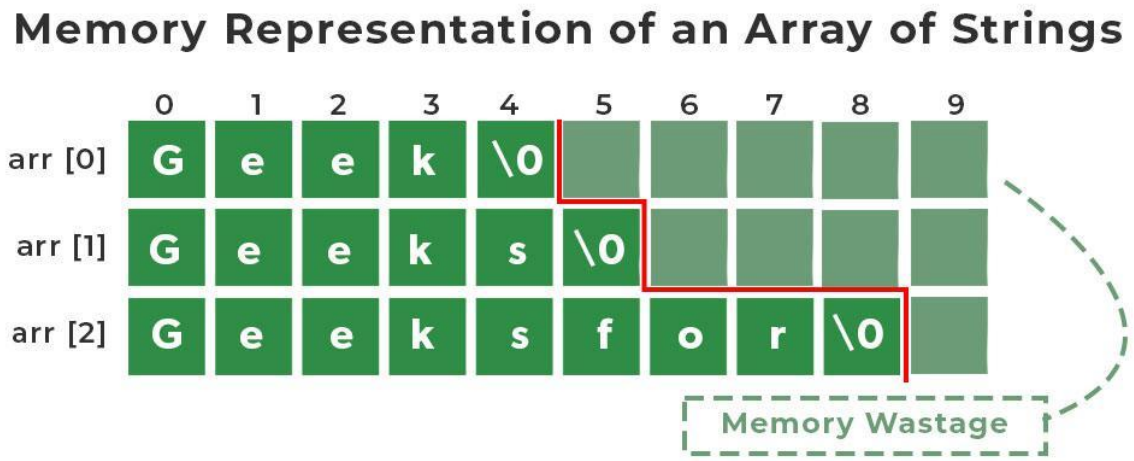
Geeksfor

Address of arr[0]: 0x400634

Address of arr[1]: 0x400639

Address of arr[2]: 0x40063f

Here, the total memory used is the memory required for storing the strings and pointers without leaving any empty space hence, saving a lot of wasted space. We can understand this using the image shown below.



The space occupied by the array of pointers to characters is shown by solid green blocks excluding the memory required for storing the pointer while the space occupied by the array of strings includes both solid and light green blocks.

Array of Pointers to Different Types

Not only we can define the array of pointers for basic data types like int, char, float, etc. but we can also define them for derived and user-defined data types such as arrays, structures, etc. Let's consider the below example where we create an array of pointers pointing to a function for performing the different operations.

Example:

```
// C program to illustrate the use of array of pointers to

// function

#include <stdio.h>


// some basic arithmetic operations
```

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

void subtract(int a, int b) {
   printf("Difference : %d\n", a - b);
}

void multiply(int a, int b) {
   printf("Product : %d\n", a * b);
}

void divide(int a, int b) {
   printf("Quotient : %d", a / b);
}

int main() {

   int x = 50, y = 5;

   // array of pointers to function of return type int
   void (*arr[4])(int, int)
      = { &add, &subtract, &multiply, &divide };
   for (int i = 0; i < 4; i++) {
      arr[i](x, y);
   }
   return 0;
}
```

Output

Sum : 55

Difference : 45

Product : 250

Quotient : 10

Application of Array of Pointers

An array of pointers is useful in a wide range of cases. Some of these applications are listed below:

- It is most commonly used to store multiple strings.
- It is also used to implement LinkedHashMap in C and also in the Chaining technique of collision resolving in Hashing.
- It is used in sorting algorithms like bucket sort.
- It can be used with any pointer type so it is useful when we have separate declarations of multiple entities and we want to store them in a single place.

Disadvantages of Array of Pointers

The array of pointers also has its fair share of disadvantages and should be used when the advantages outweigh the disadvantages. Some of the disadvantages of the array of pointers are:

- **Higher Memory Consumption:** An array of pointers requires more memory as compared to plain arrays because of the additional space required to store pointers.
- **Complexity:** An array of pointers might be complex to use as compared to a simple array.
- **Prone to Bugs:** As we use pointers, all the bugs associated with pointers come with it so we need to handle them carefully.

Structure and Union in C

In C programming, both structures and unions are used to group different types of data under a single name, but they behave in different ways. The main difference lies in how they store data.

The below table lists the primary differences between the C structures and unions:

Parameter	Structure	Union
Definition	A structure is a user-defined data type that groups different data types into a single entity.	A union is a user-defined data type that allows storing different data types at the same memory location.

Parameter	Structure	Union
Keyword	The keyword **struct** is used to define a structure	The keyword **union** is used to define a union
Size	The size is the sum of the sizes of all members, with padding if necessary.	The size is equal to the size of the largest member, with possible padding.
Memory Allocation	Each member within a structure is allocated unique storage area of location.	Memory allocated is shared by individual members of union.
Data Overlap	No data overlap as members are independent.	Full data overlap as members shares the same memory.
Accessing Members	Individual member can be accessed at a time.	Only one member can be accessed at a time.

Structures

A **structure in C** is a collection of variables, possibly of different types, under a single name. Each member of the structure is allocated its own memory space, and the size of the structure is the sum of the sizes of all its members.

Syntax

struct *name {*
member1 definition;
member2 definition;
...
memberN definition;
};

Example:

```
#include <stdio.h>


struct Student {
   char name[50];
   int age;
   float grade;
```

```
};

int main() {

    // Create a structure variable
    struct Student s1 = {"Geek", 20, 85.5};

    // Access structure members
    printf("%s\n", s1.name);
    printf("%d\n", s1.age);
    printf("%.2f\n", s1.grade);
    printf("Size: %d bytes", sizeof(s1));

    return 0;
}
```

Output

```
Geek
20
85.50
Size: 60 bytes
```

Explanation: In this example, we create a structure Student to store a student's name, age, and grade. Each of the members (name, age, grade) is stored in its own separate memory location, and we access them individually. The size is also the size of all members combined plus structure padding.

Unions

A **union in C** is similar to a structure, but with a key difference: all members of a union share the same memory location. This means only one member of the union can store a value at any given time. The size of a union is determined by the size of its largest member.

Syntax:

union *name {*
member1 definition;
member2 definition;
...

memberN definition;
};

Example:

⇔

```
union Data {
   int i;
   double d;
   char c;
};

int main() {

   // Create a union variable
   union Data data;

   // Store an integer in the union
   data.i = 100;
   printf("%d
", data.i);

   // Store a double in the union (this will
   // overwrite the integer value)
   data.d = 99.99;
   printf("%.2f
", data.d);

   // Store a character in the union (this will
   // overwrite the double value)
   data.c = 'A';
   printf("%c
", data.c);
```

```
    printf("Size: %d", sizeof(data));
```

↔

Output

100

99.99

A

Size: 8

Similarities Between Structure and Union

Structures and unions are also similar in some aspects listed below:

- Both are user-defined data types used to store data of different types as a single unit.
- Their members can be objects of any type, including other structures and unions or arrays. A member can also consist of a bit field.
- Both structures and unions support only assignment = and sizeof operators. The two structures or unions in the assignment must have the same members and member types.
- A structure or a union can be passed by value to functions and returned by value by functions. The argument must have the same type as the function parameter. A structure or union is passed by value just like a scalar variable as a corresponding parameter.
- ‘.’ operator or selection operator, which has one of the highest precedences, is used for accessing member variables inside both the user-defined datatypes.

UNIT V

Functions & File Handling Introduction to Functions, Function Declaration and Definition, Function call Return Types and Arguments, modifying parameters inside functions using pointers, arrays as parameters. Scope and Lifetime of Variables, Basics of File Handling

C Functions

A function is a block of code that performs a specific task.

Suppose, you need to create a program to create a circle and color it. You can create two functions to solve this problem:

- create a circle function
- create a color function

Dividing a complex problem into smaller chunks makes our program easy to understand and reuse.

Types of function

There are two types of function in C programming:

- Standard library functions
- User-defined functions

Standard library functions

The standard library functions are built-in functions in C programming.

These functions are defined in header files. For example,

- The printf() is a standard library function to send formatted output to the screen (display output on the screen). This function is defined in the stdio.h header file.
  Hence, to use the printf()function, we need to include the stdio.h header file using #include <stdio.h>.
- The sqrt() function calculates the square root of a number. The function is defined in the math.h header file.

Visit standard library functions in C programming to learn more.

User-defined function

You can also create functions as per your need. Such functions created by the user are known as user-defined functions.

How user-defined function works?

```
#include <stdio.h>
void functionName()
{
   ... .. ...
   ... .. ...
}

int main()
{
   ... .. ...
   ... .. ...

   functionName();

   ... .. ...
   ... .. ...
}
```

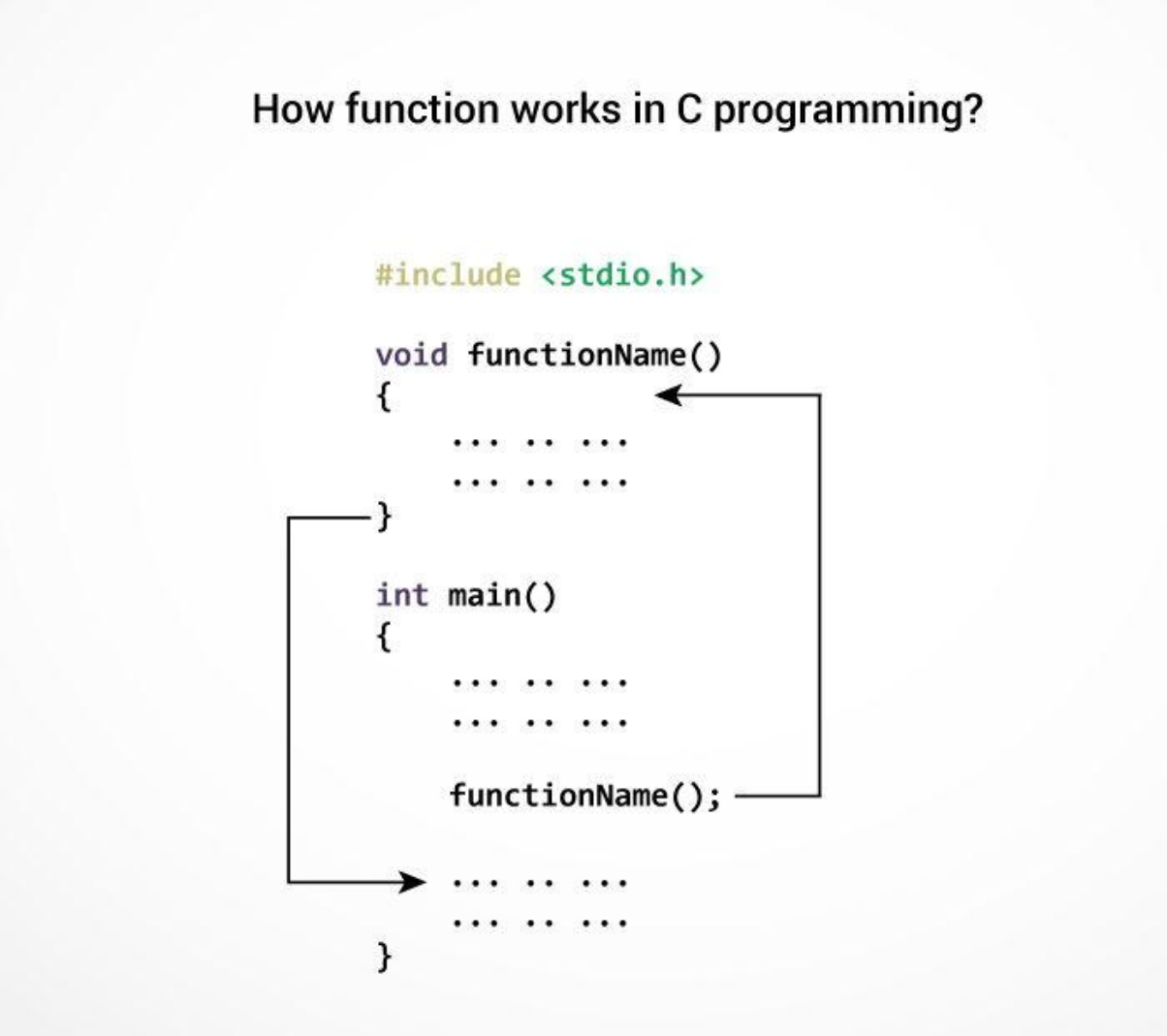
The execution of a C program begins from the main() function.

When the compiler encounters functionName();, control of the program jumps to

void functionName()

And, the compiler starts executing the codes inside functionName().

The control of the program jumps back to the main() function once code inside the function definition is executed.



Working of C Function

Note, function names are identifiers and should be unique.

This is just an overview of user-defined functions. Visit these pages to learn more on:

- User-defined Function in C programming
- Types of user-defined Functions

Advantages of user-defined function

1. The program will be easier to understand, maintain and debug.
2. Reusable codes that can be used in other programs
3. A large program can be divided into smaller modules. Hence, a large project can be divided among many programmers.

C User-defined functions

A function is a block of code that performs a specific task.

C allows you to define functions according to your need. These functions are known as user-defined functions. For example:

Suppose, you need to create a circle and color it depending upon the radius and color. You can create two functions to solve this problem:

- createCircle() function
- color() function

Example: User-defined function

Here is an example to add two integers. To perform this task, we have created an user-defined addNumbers().

```
#include <stdio.h>

int addNumbers(int a, int b);       // function prototype


int main()

{

   int n1,n2,sum;


   printf("Enters two numbers: ");

   scanf("%d %d",&n1,&n2);


   sum = addNumbers(n1, n2);      // function call

   printf("sum = %d",sum);


   return 0;

}


int addNumbers(int a, int b)       // function definition

{

   int result;

   result = a+b;

   return result;              // return statement
```

}

Function prototype

A function prototype is simply the declaration of a function that specifies function's name, parameters and return type. It doesn't contain function body.

A function prototype gives information to the compiler that the function may later be used in the program.

Syntax of function prototype

returnType functionName(type1 argument1, type2 argument2, ...);

In the above example, int addNumbers(int a, int b); is the function prototype which provides the following information to the compiler:

1. name of the function is addNumbers()
2. return type of the function is int
3. two arguments of type int are passed to the function

The function prototype is not needed if the user-defined function is defined before the main() function.

Calling a function

Control of the program is transferred to the user-defined function by calling it.

Syntax of function call

functionName(argument1, argument2, ...);

In the above example, the function call is made using addNumbers(n1, n2); statement inside the main() function.

Function definition

Function definition contains the block of code to perform a specific task. In our example, adding two numbers and returning it.

Syntax of function definition

returnType functionName(type1 argument1, type2 argument2, ...)

{

  //body of the function

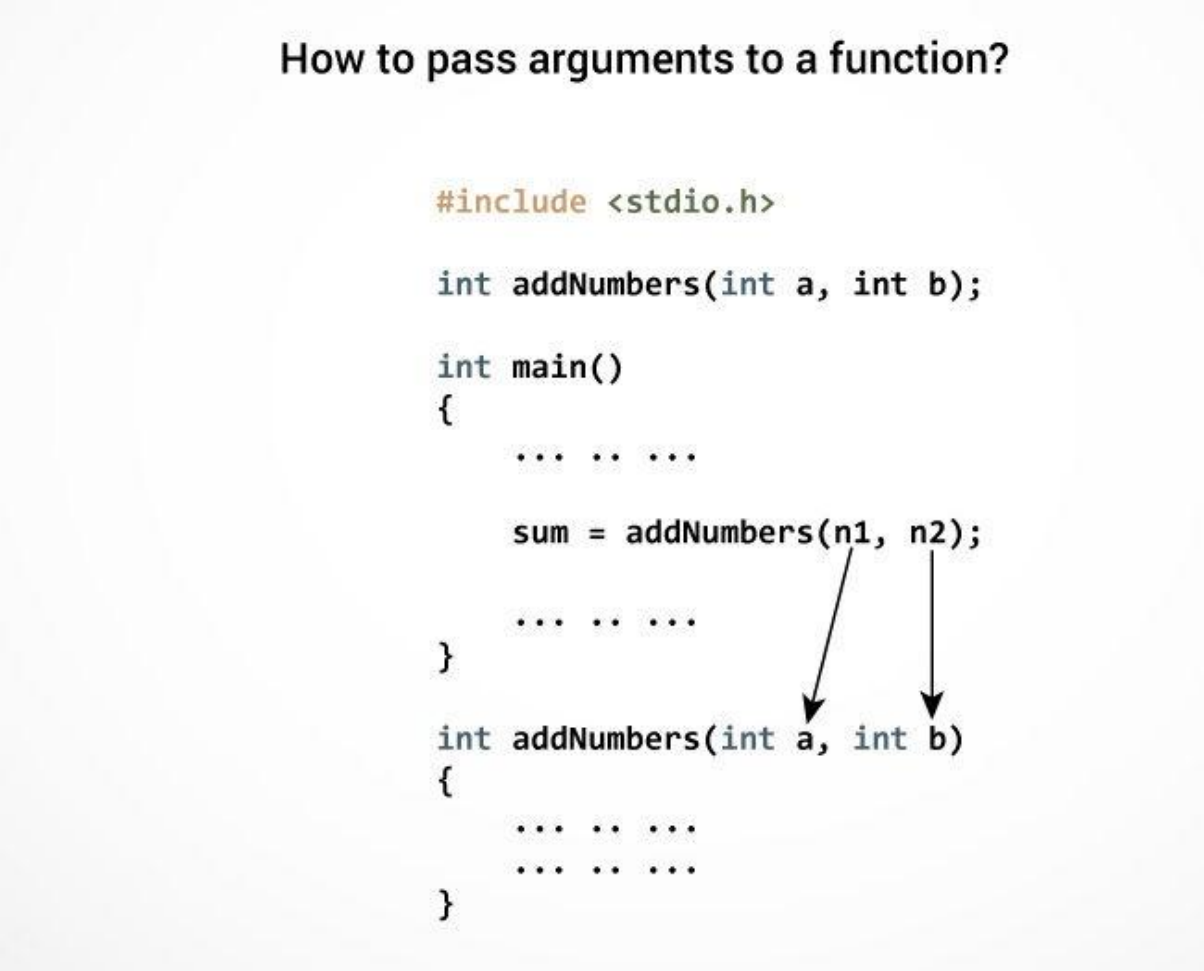
}

When a function is called, the control of the program is transferred to the function definition. And, the compiler starts executing the codes inside the body of a function.

Passing arguments to a function

In programming, argument refers to the variable passed to the function. In the above example, two variables n1 and n2 are passed during the function call.

The parameters a and b accepts the passed arguments in the function definition. These arguments are called formal parameters of the function.



Passing Argument to Function

The type of arguments passed to a function and the formal parameters must match, otherwise, the compiler will throw an error.

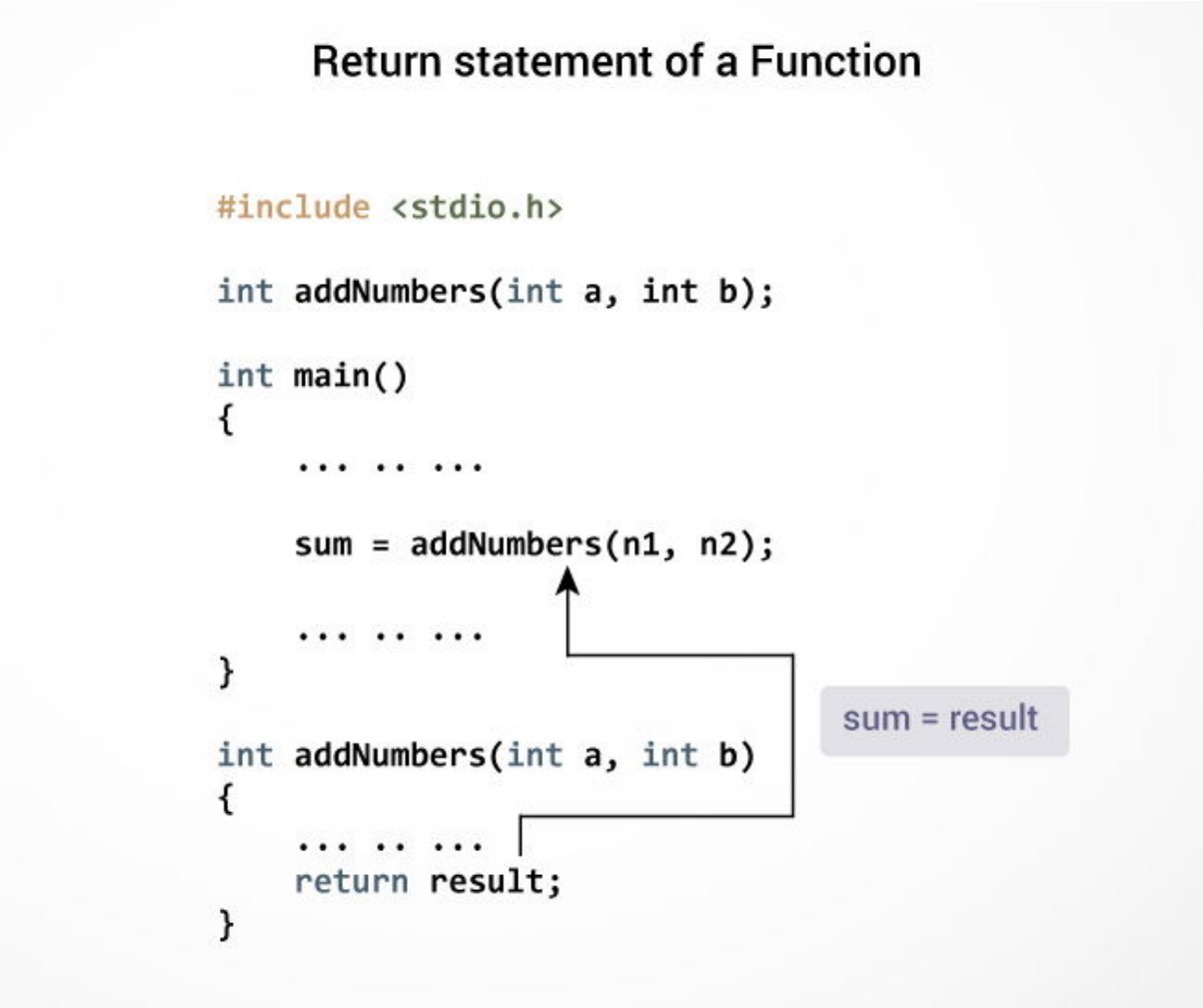
If n1 is of char type, a also should be of char type. If n2 is of float type, variable b also should be of float type.

A function can also be called without passing an argument.

Return Statement

The return statement terminates the execution of a function and returns a value to the calling function. The program control is transferred to the calling function after the return statement.

In the above example, the value of the result variable is returned to the main function.
The sum variable in the main() function is assigned this value.



Return Statement of Function

Syntax of return statement

return (expression);

For example,

return a;

return (a+b);

The type of value returned from the function and the return type specified in the function prototype and function definition must match.

Types of User-defined Functions in C Programming

These 4 programs below check whether the integer entered by the user is a prime number or not.

The output of all these programs below is the same, and we have created a user-defined function in each example. However, the approach we have taken in each example is different.

Example 1: No Argument Passed and No Return Value

```
#include <stdio.h>


void checkPrimeNumber();


int main() {
 checkPrimeNumber();    // argument is not passed
 return 0;
}


// return type is void meaning doesn't return any value
void checkPrimeNumber() {
 int n, i, flag = 0;


 printf("Enter a positive integer: ");
 scanf("%d",&n);


 // 0 and 1 are not prime numbers
 if (n == 0 || n == 1)
  flag = 1;


 for(i = 2; i <= n/2; ++i) {
  if(n%i == 0) {
   flag = 1;
   break;
  }
 }
```

```
 if (flag == 1)
   printf("%d is not a prime number.", n);
 else
   printf("%d is a prime number.", n);
}
```

The checkPrimeNumber() function takes input from the user, checks whether it is a prime number or not, and displays it on the screen.

The empty parentheses in checkPrimeNumber(); inside the main() function indicates that no argument is passed to the function.

The return type of the function is void. Hence, no value is returned from the function.

Example 2: No Arguments Passed But Returns a Value

```
#include <stdio.h>
int getInteger();

int main() {

 int n, i, flag = 0;

 // no argument is passed
 n = getInteger();

 // 0 and 1 are not prime numbers
 if (n == 0 || n == 1)
   flag = 1;

 for(i = 2; i <= n/2; ++i) {
   if(n%i == 0){
     flag = 1;
     break;
   }
```

```
 }

 if (flag == 1)
  printf("%d is not a prime number.", n);
 else
  printf("%d is a prime number.", n);

 return 0;
}

// returns integer entered by the user
int getInteger() {
 int n;

 printf("Enter a positive integer: ");
 scanf("%d",&n);

 return n;
}
```

The empty parentheses in the n = getInteger(); statement indicates that no argument is passed to the function. And, the value returned from the function is assigned to n.

Here, the getInteger() function takes input from the user and returns it. The code to check whether a number is prime or not is inside the main() function.

Example 3: Argument Passed But No Return Value

```
#include <stdio.h>
void checkPrimeAndDisplay(int n);

int main() {

 int n;
```

```
 printf("Enter a positive integer: ");
 scanf("%d",&n);

 // n is passed to the function
 checkPrimeAndDisplay(n);

 return 0;
}

// return type is void meaning doesn't return any value
void checkPrimeAndDisplay(int n) {
 int i, flag = 0;

 // 0 and 1 are not prime numbers
 if (n == 0 || n == 1)
  flag = 1;

 for(i = 2; i <= n/2; ++i) {
  if(n%i == 0){
   flag = 1;
   break;
  }
 }

 if(flag == 1)
  printf("%d is not a prime number.",n);
 else
  printf("%d is a prime number.", n);
}
```

The integer value entered by the user is passed to the checkPrimeAndDisplay() function.

Here, the checkPrimeAndDisplay() function checks whether the argument passed is a prime number or not and displays the appropriate message.

Example 4: Argument Passed and Returns a Value

```
#include <stdio.h>

int checkPrimeNumber(int n);


int main() {


 int n, flag;


 printf("Enter a positive integer: ");

 scanf("%d",&n);


 // n is passed to the checkPrimeNumber() function

 // the returned value is assigned to the flag variable

 flag = checkPrimeNumber(n);


 if(flag == 1)

  printf("%d is not a prime number",n);

 else

  printf("%d is a prime number",n);


 return 0;

}


// int is returned from the function

int checkPrimeNumber(int n) {


 // 0 and 1 are not prime numbers
```

```
 if (n == 0 || n == 1)
   return 1;

 int i;

 for(i=2; i <= n/2; ++i) {
   if(n%i == 0)
     return 1;
 }

 return 0;
}
```

The input from the user is passed to the checkPrimeNumber() function.

The checkPrimeNumber() function checks whether the passed argument is prime or not.

If the passed argument is a prime number, the function returns **0**. If the passed argument is a non-prime number, the function returns **1**. The return value is assigned to the flag variable.

Depending on whether flag is **0** or **1**, an appropriate message is printed from the main() function.

Which approach is better?

Well, it depends on the problem you are trying to solve. In this case, passing an argument and returning a value from the function (example 4) is better.

A function should perform a specific task. The checkPrimeNumber() function doesn't take input from the user nor it displays the appropriate message. It only checks whether a number is prime or not.

Function Prototype in C

In C, a **function prototype** is a statement that tells the compiler about the function's name, its return type, numbers, and data types of its parameters. Using this information, the compiler cross-checks function parameters and their data type with function definition and function call.

For example, let's create a function prototype for a function that adds two numbers which it takes as arguments and returns their sum:

```
// Not a prototype as it does not contain the number
// and type of parameters. It is only declaration
int sum();

// Valid prototype as it contains function name,
// return type, number and type of parameters
int sum(int, int);

// Also a valid prototype
int sum(int a, int b);

// Function definition inherently contains function
// prototype
int sum(int a, int b) {
        return a + b;
}
```

General Syntax

return_type name(type1, type2 ....);

where,

- **return_type :** Type of value the function returns.
- **name:** Name of the function
- **type1, type2 ....:** Type of the parameters. Specifying their names is optional.

Function prototype can be used in place of function declaration in cases where the function reference or call is present before the function definition but optional if the function definition is present before the function call in the program.

What Happens If the Function Prototype Is Missing?

When the function prototype is missing in C, the compiler generally gives two warnings:

1. Implicit Declaration Warning.
2. Incorrect return type warning (in case where the function should return non-integer value)

The first warning is understandable as the compiler cannot find the function with the given name. The other warning is due to the fact that the compiler assumes the return type to be int by default in case of missing return type(or whole prototype).

The below program demonstrate this:

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

int main(int argc, char* argv[]) {
  FILE* fp;

  fp = fopen(argv[1], "r");
  if (fp == NULL) {

    // Calling strerror() which is defined inside
    // string.h header file
    fprintf(stderr, "%s\n", strerror(errno));
    return errno;
  }

  printf("file exist\n");

  fclose(fp);

  return 0;
}
```

Output

```
solution.c: In function ‘main’:
solution.c:14:41: warning: implicit declaration of function ‘strerror’; did you mean ‘perror’? [-Wimplicit-function-declaration]
   14 |         fprintf(stderr, "%s\n", strerror(errno));
      |                                 ^~~~~~~~
      |                                 perror
solution.c:14:35: warning: format ‘%s’ expects argument of type ‘char *’, but argument 3 has type ‘int’ [-Wformat=]
```

```
14 |         fprintf(stderr, "%s\n", strerror(errno));
   |                         ~^    ~~~~~~~~~~~~~~~~
   |                          |    |
   |                          |    int
   |                          char *
   |                         %d
```

Segmentation fault (core dumped)

Explanation

The above program checks the existence of a file, provided from the command line, if a given file exists, then the program prints "file exists", otherwise it prints an appropriate error message.

Why did this program crash, instead it should show an appropriate error message. This program will work fine on x86 architecture but will crash on x86_64 architecture. Let us see what was wrong with the code. Carefully go through the program, deliberately I haven't included the prototype of the "strerror()" function. This function returns a "pointer to the character", which will print an error message which depends on the errno passed to this function.

Note that x86 architecture is an ILP-32 model, which means integers, pointers, and long are 32-bit wide, that's why the program will work correctly on this architecture. But x86_64 is the LP-64 model, which means long and pointers are 64-bit wide. *In C language, when we don't provide a prototype of a function, the compiler assumes that the function returns an integer*. In our example, we haven't included the "string.h" header file (strerror's prototype is declared in this file), that's why the compiler assumed that the function returns an integer. But its return type is a pointer to a character.

In x86_64, pointers are 64-bit wide and integers are 32-bit wide, that's why while returning from a function, the returned address gets truncated (i.e. 32-bit wide address, which is the size of integer on x86_64) which is invalid and when we try to dereference this address, the result is a segmentation fault.

Now include the "string.h" header file and check the output, the program will work correctly and gives the following output.

Benifits of Function Prototypes

Following are the major benefits of including function prototypes in your program:

- **Function Declaration Before Definition:** It allows a function to be called before its definition. In large programs, it is common to place function prototypes at the beginning of the code or in header files, enabling function calls to occur before the function's actual implementation.
- **Type Checking:** A function prototype allows the compiler to check that the correct number and type of arguments are passed to the function. If the function is called with the wrong type or number of parameters, the compiler can catch the error.

- **Code Clarity:** By declaring prototypes, you inform the programmer about the function's purpose and expected parameters, which helps in understanding the code structure.

Function Declaration vs Prototype

The terms function declaration and function prototypes are often used interchangeably but they are different in the purpose and their meaning. Following are the major differences between the function declaration and function prototype in C:

Function Declaration	Function Prototype
Function Declaration is used to tell the existence of a function.	The function prototype tells the compiler about the existence and signature of the function.
A function declaration is valid even with only function name and return type.	A function prototype is a function delcaration that provides the function's name, return type, and parameter list without including the function body.
Typically used in header files to declare functions.	Used to declare functions before their actual definitions.
Syntax: return_type function_name();	**Syntax:** return_type function_name(parameter_list);

How can I return multiple values from a function?

In C programming, a function can return only one value directly. However, C also provides several indirect methods in to return multiple values from a function. In this article, we will learn the different ways to return multiple values from a function in C.

The most straightforward method to return multiple values from a function is by using a **structure.** Let's take a look at an example:

```
#include <stdio.h>


// Structure of multiple values
```

```
struct A {
   int a;
   char c;
};

// Function that returns struct A
struct A func() {
   struct A r;
   r.a = 10;
   r.c = 'z';

   // Return the struct
   return r;
}

int main() {

   // Storing the returned structure
   struct A res = func();

   printf("a = %d, c = %c\n", res.a, res.c);
   return 0;
}
```

Output

a = 10, c = z

Explanation: In the above program, structure **A** with multiple members is used returning multiple values from the function **func().** In main() function, the returned structure is stored, and its values are printed.

C also have some other methods to return multiple value from a function. They are as follows:

Using Arrays

If the multiple values to be returned are of same type, then the **array** of these values can be returned. But it is to be noted that the array should be either declared as static or should be dynamically allocated so that they don't get destroyed when the function ends.

```
#include <stdio.h>


int* func() {


  // Creating static array

  static int arr[2] = {10, 20};


  // Returning multiple values using static array

  return arr;

}

int main() {


  // Store the returened array

  int* arr = func();

  printf("%d %d", arr[0], arr[1]);

  return 0;

}
```

Output

10 20

Explanation: In this case, an array values[] is passed to the function getValues, which assigns values to the array. Since arrays in C are passed by reference, the changes made inside the function affect the original array.

Using Pointers

This method is technically not for returning but writing the result of the function in the address (pointers) already passed as parameters. It is an alternative to returning structures or arrays.

```
#include <stdio.h>



void foo(int* a, char* c) {
```

```
    // Writing data to passed addresses
    *a = 20;
    *c = 'z';
}

int main() {
    int a;
    char c;
    foo(&a, &c);

    printf("a = %d, c = %d", a, c);
    return 0;
}
```

Output

a = 20, c = 122

We can also pass arrays in place of seperate variable addresses or even use global variable for storing the results.

Difference between Argument and Parameter in C with Examples

Argument

An **argument** is referred to the values that are passed within a function when the function is called. These values are generally the source of the function that require the arguments during the process of execution. These values are assigned to the variables in the definition of the function that is called. The type of the values passed in the function is the same as that of the variables defined in the function definition. These are also called **Actual arguments** or **Actual Parameters**. **Example:** Suppose a sum() function is needed to be called with two numbers to add. These two numbers are referred to as the arguments and are passed to the sum() when it called from somewhere else.

CC++

```
// C code to illustrate Arguments

#include <stdio.h>
```

```
// sum: Function definition
int sum(int a, int b)
{
   // returning the addition
   return a + b;
}

// Driver code
int main()
{
   int num1 = 10, num2 = 20, res;

   // sum() is called with
   // num1 & num2 as ARGUMENTS.
   res = sum(num1, num2);

   // Displaying the result
   printf("The summation is %d", res);
   return 0;
}
```

Output:

The summation is 30

Parameters

The parameter is referred to as the variables that are defined during a function declaration or definition. These variables are used to receive the arguments that are passed during a function call. These parameters within the function prototype are used during the execution of the function for which it is defined. These are also called Formal arguments or Formal Parameters. **Example:** Suppose a Mult() function is needed to be defined to multiply two numbers. These two numbers are referred to as the parameters and are defined while defining the function Mult().

CC++

```
// C code to illustrate Parameters
```

```
#include <stdio.h>

// Mult: Function definition
// a and b are the PARAMETERS
int Mult(int a, int b)
{
   // returning the multiplication
   return a * b;
}

// Driver code
int main()
{
   int num1 = 10, num2 = 20, res;

   // Mult() is called with
   // num1 & num2 as ARGUMENTS.
   res = Mult(num1, num2);

   // Displaying the result
   printf("The multiplication is %d", res);
   return 0;
}
```

Output:

The multiplication is 200

Difference between Argument and Parameter

Argument	Parameter
When a function is called, the values that are passed during the call are called as arguments.	The values which are defined at the time of the function prototype or definition of the function are called as parameters.

Argument	Parameter
These are used in function call statement to send value from the calling function to the receiving function.	These are used in function header of the called function to receive the value from the arguments.
During the time of call each argument is always assigned to the parameter in the function definition.	Parameters are local variables which are assigned value of the arguments when the function is called.
They are also called Actual Parameters	They are also called Formal Parameters
Example: int num = 20; Call(num) *// num is argument*	**Example:** int Call(int rnum) { printf("the num is %d", rnum); } *// rnum is parameter*

Pass Array to Functions in C

Passing an array to a function allows the function to directly access and modify the original array. In this article, we will learn how to pass arrays to functions in C.

In C, **arrays are always passed to function as pointers.** They cannot be passed by value because of the array decay due to which, whenever array is passed to a function, it decays into a pointer to its first element. However, there can be different syntax of passing the arrays as pointers.

The easiest way to pass array to a function is by defining the parameter as the **undefined sized array**. Let's take a look at an example:

```
#include <stdio.h>


// Array passed as an array without the size of its
```

```
// dimension
void printArr(int arr[], int n) {
   for (int i = 0; i < n; i++)
      printf("%d ", arr[i]);
}

int main() {
   int arr[] = {1, 2, 3, 4, 5};

   // Pass array to function
   printArr(arr, 5);

   return 0;
}
```

Output

1 2 3 4 5

Explanation: In this example, we pass an integer array **arr** to the function **printArr()**. In the function prototype, we define the array as the array of integers with no size information.

Here, you may also notice that we have passed the size of the array as second parameter to the function. It **is recommended to pass the size of the array to the function as another parameter**, otherwise we won't know how many elements to process.

The other ways of passing arrays as pointers include:

Passing as Sized Array

Similar to the above method, we can also define the size of the array while passing it to the function. But it still will be treated as pointer in the function.

```
#include <stdio.h>

// Array passed as an array with the size of its
// dimension
void printArr(int arr[5], int n) {
   for (int i = 0; i < n; i++)
      printf("%d ", arr[i]);
```

```
}

int main() {
    int arr[] = {1, 2, 3, 4, 5};

    // Pass array to function
    printArr(arr, 5);

    return 0;
}
```

Output

1 2 3 4 5

Explanation: Here, we pass the array **arr** and its size **size** to the function **printArray**. The function then prints all elements of the array based on the given size.

Passing Array as Pointer Notation

Instead of using array notation arr[] in the function parameter, we can directly use pointer notation int *arr. All are equivalent, as arrays in C are treated as pointers to the first element of the array. This method is more flexible when working with dynamically allocated arrays.

```
#include <stdio.h>

// Array passed as an array with the size of its
// dimension
void printArr(int* arr, int n) {
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
}

int main() {
    int arr[] = {1, 2, 3, 4, 5};

    // Pass array to function
```

```
    printArr(arr, 5);


    return 0;
}
```

Output

1 2 3 4 5

Passing Pointers to Functions in C

Passing the pointers to the function means the memory location of the variables is passed to the parameters in the function, and then the operations are performed. The function definition accepts these addresses using pointers, addresses are stored using pointers.

Arguments Passing without pointer

When we pass arguments without pointers the changes made by the function would be done to the local variables of the function.

Below is the C program to pass arguments to function without a pointer:

```
// C program to swap two values
//  without passing pointer to
// swap function.
#include <stdio.h>


void swap(int a, int b)
{
  int temp = a;
  a = b;
  b = temp;
}


// Driver code
int main()
{
  int a = 10, b = 20;
  swap(a, b);
```

```
 printf("Values after swap function are: %d, %d",
       a, b);
 return 0;
}
```

Output

Values after swap function are: 10, 20

Arguments Passing with pointers

A pointer to a function is passed in this example. As an argument, a pointer is passed instead of a variable and its address is passed instead of its value. As a result, any change made by the function using the pointer is permanently stored at the address of the passed variable. In C, this is referred to as call by reference.

Below is the C program to pass arguments to function with pointers:

```
// C program to swap two values
// without passing pointer to
// swap function.
#include <stdio.h>

void swap(int* a, int* b)
{
 int temp;
 temp = *a;
 *a = *b;
 *b = temp;
}

// Driver code
int main()
{
 int a = 10, b = 20;
 printf("Values before swap function are: %d, %d\n",
       a, b);
```

```
 swap(&a, &b);
 printf("Values after swap function are: %d, %d",
       a, b);
 return 0;
}
```

Output

Values before swap function are: 10, 20

Values after swap function are: 20, 10

Scope and Lifetime of Variables

In C, the scope and lifetime of a variable determine where a variable can be accessed in the code and how long it retains its value in memory. Let's break these down into different types of scopes and storage classes, each with different characteristics.

Types of Variable Scope

1. Local Scope: Variables declared inside a function or block.
2. Global Scope: Variables declared outside all functions.
3. Block Scope: Variables declared within a specific block, such as inside loops or conditionals.
4. File Scope: Variables declared at the file level (outside functions) and limited to the file when marked as static.

Lifetime of Variables

- The lifetime refers to how long the variable retains its value in memory.
- Variables can have automatic, static, or dynamic lifetimes depending on where and how they're declared.

Let's look at examples that illustrate scope and lifetime.

Example 1: Local Scope and Automatic Lifetime

Local variables declared within a function have local scope and automatic lifetime. This means they are created when the function is called and destroyed when the function exits.

```
#include <stdio.h>


void exampleFunction() {
   int localVar = 10;  // Local to exampleFunction
   printf("Local variable: %d\n", localVar);
```

```
}

int main() {
   exampleFunction();
   // printf("%d", localVar);  // Error: localVar not accessible here
   return 0;
}
```

In this example, localVar exists only within exampleFunction. When exampleFunction is called, localVar is created, and when the function exits, it is destroyed.

Example 2: Global Scope and Lifetime

Global variables are declared outside of any function and have global scope. They are accessible from any function in the same file and have a static lifetime, meaning they exist for the entire runtime of the program.

```
#include <stdio.h>

int globalVar = 20;  // Global variable

void display() {
   printf("Global variable: %d\n", globalVar);
}

int main() {
   printf("Accessing global variable in main: %d\n", globalVar);
   display();  // Also accesses globalVar
   return 0;
}
```

In this case, globalVar is accessible both in main and display functions and remains in memory for the program's lifetime.

Example 3: Block Scope and Limited Lifetime

Block scope applies to variables declared within specific blocks like loops, conditionals, or other code blocks.

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

```
int main() {
   for (int i = 0; i < 3; i++) {  // i has block scope
      printf("i in loop: %d\n", i);
   }
   // printf("%d", i);  // Error: i is not accessible here
   return 0;
}
```

Here, the variable i has block scope, limited to the for loop. It is created when the loop starts and destroyed when the loop ends.

Example 4: Static Local Variable

A static local variable has a local scope (accessible only within the function) but a static lifetime (it retains its value between function calls).

```
#include <stdio.h>

void staticExample() {
   static int count = 0;  // Static local variable
   count++;
   printf("Static variable count: %d\n", count);
}

int main() {
   staticExample();  // Output: 1
   staticExample();  // Output: 2
   staticExample();  // Output: 3
   return 0;
}
```

Here, count is a local variable with static storage, so it retains its value between calls to staticExample.

Example 5: extern Keyword for Global Variables Across Files

The extern keyword is used to declare a global variable in another file, allowing access across multiple files.

File 1 (file1.c):

```
#include <stdio.h>

int globalVar = 100;  // Global variable definition

void display() {
   printf("Global variable in file1: %d\n", globalVar);
}
```

File 2 (file2.c):

```
#include <stdio.h>

extern int globalVar;  // Declaration of global variable in another file

int main() {
   printf("Accessing globalVar from file2: %d\n", globalVar);
   return 0;
}
```

In file2.c, we declare globalVar with extern, indicating that it is defined in another file. This allows file2.c to access globalVar from file1.c.

Summary Table

Scope	Location	Lifetime	Accessibility
Local	Inside a function/block	Automatic	Only within the function/block
Global	Outside any function	Static	Accessible by any function in the file
Block	Inside a specific block	Automatic	Only within the block
Static Local	Inside a function with static	Static	Only within the function, retains value between calls

Scope	Location	Lifetime	Accessibility
Extern (Global)	Outside any function, declared extern in other files	Static	Accessible in other files with extern

Basics of File Handling in C

File handling in C is the process in which we create, open, read, write, and close operations on a file. C language provides different functions such as **fopen()**, **fwrite()**, **fread()**, **fseek()**, **fprintf()**, etc. to perform input, output, and many different C file operations in our program.

Need of File Handling in C

So far, the operations in C program are done on a prompt/terminal in which the data is only stored in the temporary memory (RAM). This data is deleted when the program is closed. But in the software industry, most programs are written to store the information fetched from the program. The use of file handling is exactly what the situation calls for.

File handling allows us to read and write data on files stored in the secondary memory such as hard disk from our C program.

C File Operations

C language provides the following different operations that we can perform on a file from our C program:

1. **Creating a new file.**
2. **Opening an existing file.**
3. **Reading from file.**
4. **Writing to a file.**
5. **Moving to a specific location in a file.**
6. **Closing a file.**

Components in C File Handling

Before we move on to the file handling, we need to understand a few concepts that are essential in file handling.

File

A file is a container of data. It can be classified into two types based on the way the file stores the data. They are as follows:

1. **Text Files**: A text file contains data in the **form of ASCII characters** and is generally used to store a stream of characters.
    - Each line in a text file ends with a new line character ('\n').
    - It can be read or written by any text editor.
    - They are generally stored with **.txt** file extension.
    - Text files can also be used to store the source code.
2. **Binary Files**: A binary file contains data in **binary form (i.e. 0's and 1's)** instead of ASCII characters. They contain data that is stored in a similar manner to how it is stored in the main memory.
    - The binary files can be created only from within a program and their contents can only be read by a program.
    - More secure as they are not easily readable.
    - They are generally stored with **.bin** file extension.

File Pointer

A file pointer is a reference to a particular position in the opened file. It is used in file handling to perform all file operations such as read, write, close, etc. We use the **FILE** macro to declare the file pointer variable. The FILE macro is defined inside **<stdio.h>** header file.

FILE* pointer_name;

File Pointer is used in almost all the file operations in C.

File Operation Functions

The file operations are performed by using the functions provided as the part of file handling API of C language. Following is the list of commonly used functions:

File operation	Declaration & Description
fopen() - To open a file	Declaration: FILE *fopen (const char *filename, const char *mode) fopen() function is used to open a file to perform operations such as reading, writing etc. In a C program, we declare a file pointer and use fopen() as below. fopen() function creates a new file if the mentioned file name does not exist. FILE *fp; fp=fopen ("filename", "'mode"); Where, fp – file pointer to the data type "FILE". filename – the actual file name with full path of the file. mode – refers to the operation that will be performed on the file. Example: r, w, a, r+, w+ and a+. Please refer below the description for these mode of operations.
fclose() - To close a file	Declaration: int fclose(FILE *fp); fclose() function closes the file that is being pointed by file pointer fp. In a C program, we close a file as below. fclose (fp);
fgets() - To read a file	Declaration: char *fgets(char *string, int n, FILE *fp) fgets function is used to read a file line by line. In a C program, we use fgets function as below. fgets (buffer, size, fp); where, buffer – buffer to put the data in. size – size of the buffer fp – file pointer
fprintf() - To write into a file	Declaration: int fprintf(FILE *fp, const char *format, ...);fprintf() function writes string into a file pointed by fp. In a C program, we write string into a file as below. fprintf (fp, "some data"); or fprintf (fp, "text %d", variable_name);

Open a File in C

For opening a file in C, the **fopen()** function is used with the filename or file path along with the required **access modes**.

Syntax:

FILE* fopen(*file_name, *access_mode);

Parameters

- ***file_name:*** name of the file when present in the same directory as the source file. Otherwise, full path.
- ***access_mode:*** Specifies for what operation the file is being opened.

Return Value

- If the file is opened successfully, returns a file pointer to it.
- If the file is not opened, then returns NULL.

File Opening Modes

File opening modes or access modes specify the allowed operations on the file to be opened. They are passed as an argument to the **fopen()** function. Some of the commonly used file access modes are listed below:

Opening Modes	Description
r	Searches file. If the file is opened successfully fopen( ) loads it into memory and sets up a pointer that points to the first character in it. If the file cannot be opened fopen( ) returns NULL.
rb	Open for reading in binary mode. If the file does not exist, fopen( ) returns NULL.
w	Open for writing in text mode. If the file exists, its contents are overwritten. If the file doesn't exist, a new file is created. Returns NULL, if unable to open the file.
wb	Open for writing in binary mode. If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a	Searches file. If the file is opened successfully fopen( ) loads it into memory and sets up a pointer that points to the last character in it. It opens only in the append mode. If the file doesn't exist, a new file is created. Returns NULL, if unable to open the file.
ab	Open for append in binary mode. Data is added to the end of the file. If the file does not exist, it will be created.
r+	Searches file. It is opened successfully fopen( ) loads it into memory and sets up a pointer that points to the first character in it. Returns NULL, if unable to open the file.
rb+	Open for both reading and writing in binary mode. If the file does not exist, fopen( ) returns NULL.
w+	Searches file. If the file exists, its contents are overwritten. If the file doesn't exist a new file is created. Returns NULL, if unable to open the file.

Opening Modes	Description
wb+	Open for both reading and writing in binary mode. If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a+	Searches file. If the file is opened successfully fopen( ) loads it into memory and sets up a pointer that points to the last character in it. It opens the file in both reading and append mode. If the file doesn't exist, a new file is created. Returns NULL, if unable to open the file.
ab+	Open for both reading and appending in binary mode. If the file does not exist, it will be created.

As given above, if you want to perform operations on a binary file, then you have to append 'b' at the last. For example, instead of "w", you have to use "wb", instead of "a+" you have to use "a+b".

Example:

```
#include <stdio.h>

#include <stdlib.h>


int main() {


  // File pointer to store the

  // value returned by fopen

  FILE* fptr;


  // Opening the file in read mode

  fptr = fopen("filename.txt", "r");


  // checking if the file is

  // opened successfully

  if (fptr == NULL) {

    printf("The file is not opened.");
```

```
  }
  return 0;
}
```

Output

The file is not opened.

The file is not opened because it does not exist in the source directory. But the fopen() function is also capable of creating a file if it does not exist.

Note: *If is essential to check for NULL values that might be returned by the fopen() function to avoid any errors*.

Create a File in C

The fopen() function can not only open a file but also can create a file if it does not exist already. For that, we have to use the modes that allow the creation of a file if not found such as **w, w+, wb, wb+, a, a+, ab, and ab+**.

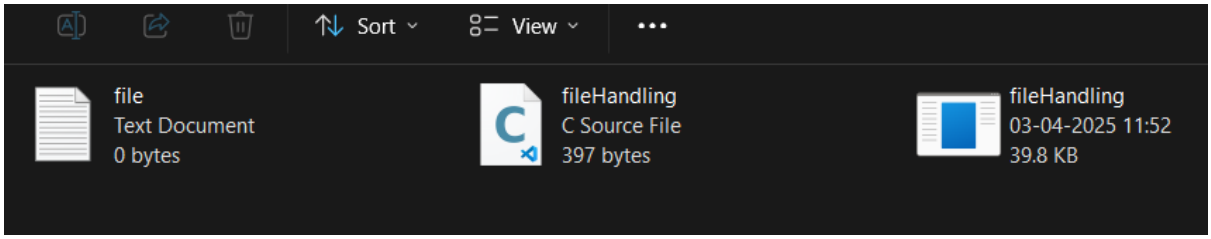
Example:

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

int main() {

  // File pointer
  FILE* fptr;

  // Creating file using fopen()
  // with access mode "w"
  fptr = fopen("file.txt", "w");

  // checking if the file is created
  if (fptr == NULL)
    printf("The file is not opened.");
  else
    printf("The file is created Successfully.");
  return 0;
```

}

Output

The file is created Successfully.



File Created

Write to a File

The file write operations can be performed by the functions fprintf() and fputs(). C programming also provides some other functions that can be used to write data to a file such as:

Function	Description
fprintf()	Similar to printf(), this function uses formatted string and variable arguments list to print output to the file.
fputs()	Prints the whole line in the file and a newline at the end.
fputc()	Prints a single character into the file.
fputw()	Prints a number to the file.
fwrite()	This function writes the specified number of bytes to the binary file.

Example:

```
#include <stdio.h>

#include <stdlib.h>


int main() {


  // File pointer

  FILE* fptr;
```

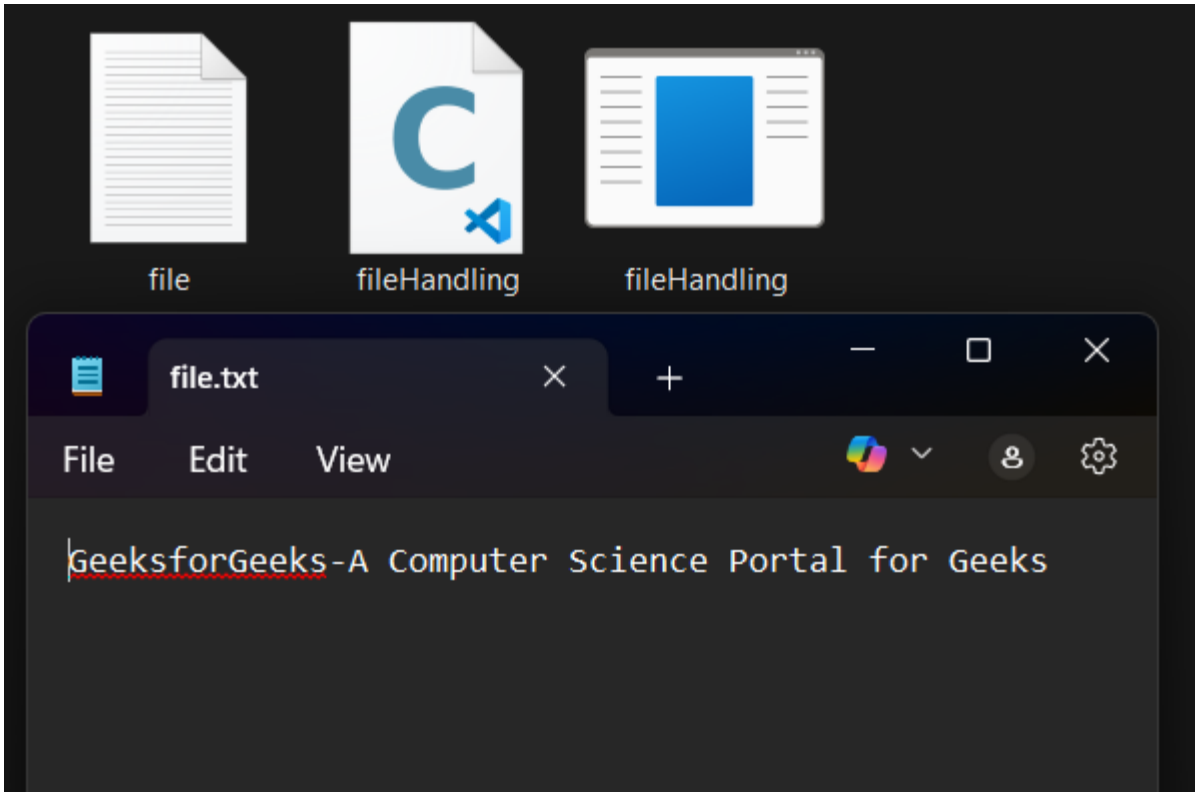
```
    // Get the data to be written in file
    char data[50] = "GeeksforGeeks-A Computer "
                    "Science Portal for Geeks";

    // Creating file using fopen()
    // with access mode "w"
    fptr = fopen("file.txt", "w");

    // Checking if the file is created
    if (fptr == NULL)
        printf("The file is not opened.");
    else{
        printf("The file is now opened.\n");
        fputs(data, fptr);
        fputs("\n", fptr);

        // Closing the file using fclose()
        fclose(fptr);
        printf("Data successfully written in file "
               "file.txt\n");
        printf("The file is now closed.");
    }
    return 0;
}
```

Output

The file is now opened.

Data successfully written in file file.txt

The file is now closed.



Reading From a File

The file read operation in C can be performed using functions **fscanf()** or **fgets()**. Both the functions performed the same operations as that of **scanf()** and gets but with an additional parameter, the file pointer. There are also other functions we can use to read from a file. Such functions are listed below:

Function	Description
fscanf()	Use formatted string and variable arguments list to take input from a file.
fgets()	Input the whole line from the file.
fgetc()	Reads a single character from the file.
fgetw()	Reads a number from a file.
fread()	Reads the specified bytes of data from a binary file.

Example:

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

```
#include <string.h>

int main() {
   FILE* fptr;

   // Declare the character array
   // for the data to be read from file
   char data[50];
   fptr = fopen("file.txt", "r");

   if (fptr == NULL) {
      printf("file.txt file failed to open.");
   }
   else {

      printf("The file is now opened.\n");

      // Read the data from the file
      // using fgets() method
      while (fgets(data, 50, fptr)
            != NULL) {

         // Print the data
         printf("%s", data);
      }

      // Closing the file using fclose()
      fclose(fptr);
   }
   return 0;
}
```

Output

The file is now opened.
GeeksforGeeks-A Computer Science Portal for Geeks

The getc() and some other file reading functions return **EOF** (**E**nd **O**f **F**ile) when they reach the end of the file while reading. EOF indicates the end of the file, and its value is implementation-defined. Reading more after EOF results in undefined error so, it is always recommended to check for EOF while reading a file.

Note: *One thing to note here is that after reading a particular part of the file, the file pointer will be automatically moved to the end of the last read character.*

Closing a File

The **fclose()** function is used to close the file. After successful file operations, you must always close a file to remove it from the memory.

Syntax:

fclose(file_pointer);

Move File Pointer

File pointer generally points to the position according to the mode or last read/write operation. We can manually move this pointer to any position in the file using **fseek()** function.

Syntax:

fseek(fptr, offset, pos);

where, ***pos*** is the position from where offset is counted and ***offset*** is the number of positions to shift from pos (it can be negative or positive)**.**

Example:

While writing to a file opened in **rw+** mode, the file pointer moves to the end of the file. In case where we want to replace a word, then first we have to move the file pointer to the position where that word starts.

```
#include <stdio.h>

#include <stdlib.h>


int main() {


  // File pointer

  FILE* fptr;


  // Get the data to be written in file
```

```
    char data[50] = "GeeksforGeeks-A Computer "
                    "Science Portal for Geeks";

    // Creating file using fopen()
    // with access mode "w"
    fptr = fopen("file.txt", "w");

    // Checking if the file is created
    if (fptr == NULL)
        printf("The file is not opened.");
    else{
        printf("The file is now opened.\n");
        fputs(data, fptr);
        fputs("\n", fptr);
        fseek(fptr, -6, SEEK_END);

        fputs("GeeksforGeeks", fptr);

        // Closing the file using fclose()
        fclose(fptr);
        printf("Data successfully written in file "
               "file.txt\n");
        printf("The file is now closed.");
    }
    return 0;
}
```

Output

The file is now opened.

Data successfully written in file file.txt

The file is now closed.

Now, imagine you want to read this file after writing. We can use **fseek()** here too, but these is one more function specifically for this purpose which is **rewind().**

Read and Write in a Binary File

Till now, we have only discussed text file operations. The operations on a binary file are similar to text file operations with little difference.

Opening a Binary File

To open a file in binary mode, we use the **rb, rb+, ab, ab+, wb, and wb+** access mode in the **fopen()** function. We also use the **.bin** file extension in the binary filename.

Example:

```
fptr = fopen("filename.bin", "rb");
```

Write to a Binary File

We use fwrite() function to write data to a binary file. The data is written to the binary file in the form of bits (0's and 1's).

Syntax of fwrite()

```
fwrite(ptr, size, nmemb, file_pointer);
```

Parameters:

- **ptr:** pointer to the block of memory to be written.
- **size:** size of each element to be written (in bytes).
- **nmemb:** number of elements.
- **file_pointer**: FILE pointer to the output file stream.

Return Value:

- Number of objects written.

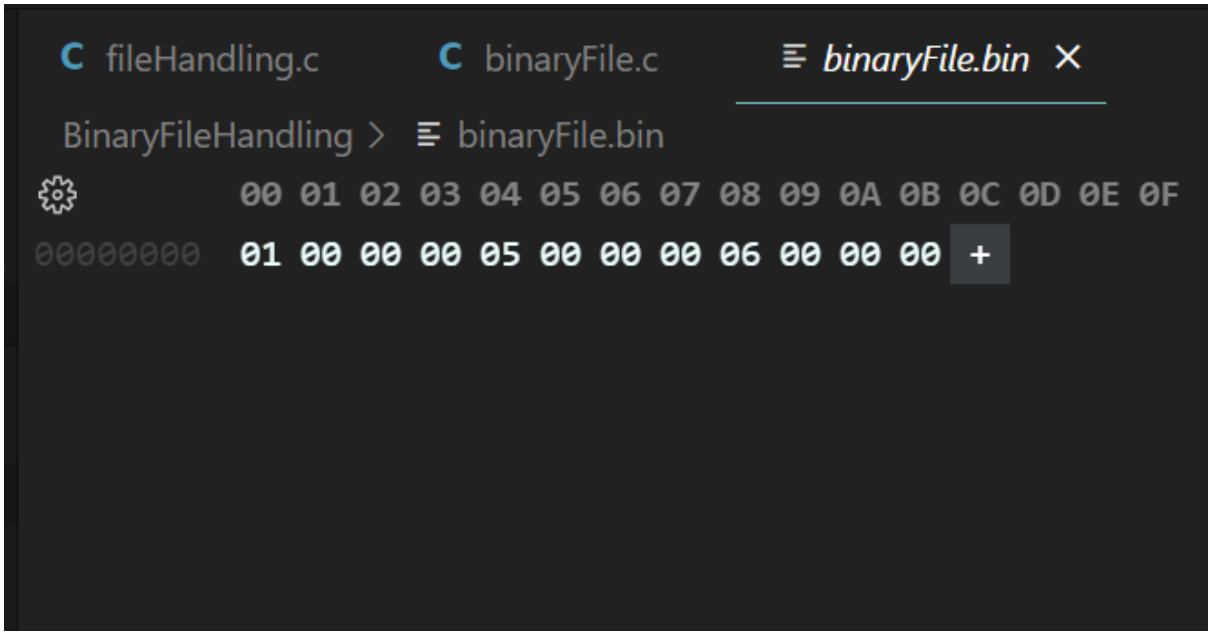
Example: Program to write to a Binary file using fwrite()

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

struct threeNum {
   int n1, n2, n3;
};

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

```
    // Structure variable declared.
    struct threeNum num;
    FILE* fptr;
    fptr = fopen("binaryFile.bin", "wb");
    int flag = 0;
    num.n1 = n;
    num.n2 = 5 * n;
    num.n3 = 5 * n + 1;

    // Write the Structure data
    // to binary file.
    flag = fwrite(&num, sizeof(struct threeNum), 1,
                  fptr);

    // Checking if the data is written.
    if (!flag)
      printf("Write Operation Failure");
    else
      printf("Write Operation Successful");
    fclose(fptr);
    return 0;
}
```

Output

Write Operation Successful



Data Written in Binary File

Reading from Binary File

The fread() function can be used to read data from a binary file in C. The data is read from the file in the same form as it is stored i.e. binary form.

Syntax:

fread(ptr, size, nmemb, file_pointer);

Parameters:

- **ptr:** pointer to the block of memory to read.
- **size:** the size of each element to read (in bytes).
- **nmemb:** number of elements.
- **file_pointer**: FILE pointer to the input file stream.

Return Value:

- Number of objects read.

Example: Program to Read from a binary file using fread()

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

// Structure that store
// binary file data
struct threeNum {
   int n1, n2, n3;
```

```
};

int main() {
   int n;
   struct threeNum num;
   FILE* fptr;
   fptr = fopen("binaryFile.bin", "rb");

   // Read the data from binary
   // file and print that data
   fread(&num, sizeof(struct threeNum), 1, fptr);
   printf("n1: %d\tn2: %d\tn3: %d\n", num.n1, num.n2,
       num.n3);
   fclose(fptr);
   return 0;
}
```

Output

n1: 1 n2: 5 n3: 6

More Functions for C File Operations

The following table lists some more functions that can be used to perform file operations or assist in performing them.

Functions	Description
fopen()	It is used to create a file or to open a file.
fclose()	It is used to close a file.
fgets()	It is used to read a file.
fprintf()	It is used to write blocks of data into a file.

Functions	Description
fscanf()	It is used to read blocks of data from a file.
getc()	It is used to read a single character to a file.
putc()	It is used to write a single character to a file.
fseek()	It is used to set the position of a file pointer to a mentioned location.
ftell()	It is used to return the current position of a file pointer.
rewind()	It is used to set the file pointer to the beginning of a file.
putw()	It is used to write an integer to a file.
getw()	It is used to read an integer from a file.

What is return type of getchar(), fgetc() and getc() ?

In C, **getchar(), fgetc(), and getc()** all are the functions used for reading characters from input buffer. This buffer is standard input buffer for getchar() and can be any specified file for getc() and fgetc(). In this article, we will learn about the return type of these functions and why it matters in C.

Return Type of getchar()

The return type of getchar() is **int**.

- Returns the ASCII value of the character as an **int.**
- If an error occurs or the end of the input is reached, returns the special constant EOF (End Of File).

Return Type of getc()

The return type of getc() is **int.**

- Returns the ASCII value of the character as an **int.**
- If it encounters the end of the file or an error, returns EOF (End Of File).

Return Type of fgetc()

The return type of fgetc() is **int.**

- Returns the ASCII value of the character as an **int.**
- If it encounters the end of the file or an error, returns EOF (End Of File).

Why the knowledge of return type matters?

As all these functions reads a single character, it is often assumed by the programmers that the return type of these functions must be of char type. But these functions either return the ASCII value or EOF which is typically defined as -1 is most implementations.

So, the return value of these functions is assigned to char (which can be an unsigned short int in some platforms), the EOF value may get converted into 2's complement leading to undefined error or infinite loop.

The below example demonstrates how it can happen:

```
#include <stdio.h>


int main() {
  FILE *fptr = fopen("file.txt", "r");
  if (!fptr) {
    perror("File opening failed");
    return 1;
  }

      // Using char to store data
  char ch;

      // Print each character
  while ((ch = fgetc(fptr)) != EOF) {
    printf("%c", ch);
  }

  fclose(fptr);
  return 0;
}
```

The above program may run well in most of the platforms, where char is signed short int. But in others, it may lead to following problems:

- **Incorrect Handling of EOF**: If EOF is returned and stored in a char variable, it might not be recognized correctly because EOF is -1, which could be misinterpreted as a valid character in the range of char.
- **Infinite Loo**p: If EOF is misinterpreted, the condition (ch != EOF) might never evaluate to false, resulting in an infinite loop.

That is why, **it is recommended to store the value returned by these functions** into an int variable and treat it as char or typecast it after verification.

What is the difference between printf, sprintf and fprintf?

The **printf()** function is used as a standard method for output operations and C also provides some different versions of this function such as **sprintf()** and **fprintf().** These functions are also used for output operations but in different contexts.

The below table lists the primary differences between the printf, sprintf and fprintf in C:

Function	Purpose	Output Location	Syntax	Use Case
printf()	Prints formatted output.	Standard output (console).	printf(format, arguments);	Use when you want to display formatted text to the console, such as for debugging.
sprintf()	Writes formatted output to a string.	A character array (string).	sprintf(buffer, format, arguments);	Use when you need to store formatted data in a string, such as for custom error messages.
fprintf()	Prints formatted output to a specific stream.	A file or stream specified by FILE*	fprintf(stream, format, arguments);	Use when you need to write formatted data to a file or another stream, such as for logging.

Let's see each of them in detail one by one.

printf()

The **printf function** is used to print formatted output to the standard output, typically the console or terminal.

Example

```
#include <stdio.h>


int main() {
   int n = 42;


   // Print integer and floating-point
   // number to the console
   printf("Number is %d", n);


   return 0;
}
```

Output

```
Number is 42
```

Explanation: The above program demonstrates how to print to the console using the **printf** function in C by passing it inside a formatted string.

sprintf

The **sprintf function** is similar to printf, but instead of printing to the console, it stores the formatted string in a character array (string). This is useful when you need to generate a string dynamically for further processing.

Example

```
#include <stdio.h>


int main() {
   int n = 42;
   char res[50];


   // Store formatted output in a string
   // using sprintf
   sprintf(res, "Number is %d.", n);
```

```
    // Print the string stored in result
    printf("%s", res);

    return 0;
}
```

Output

Book is 42.

Explanation: The **sprintf** function is used to store the formatted string in the **res** array, which is then printed using **printf.**

fprintf

The **fprintf function** is used to print formatted output to a file or any other stream, such as a printer or a log file. It works similarly to printf, but the output is directed to a file specified by the FILE pointer.

Example

```
#include <stdio.h>

int main() {
    int n = 42;
    FILE *file = fopen("file.txt", "w");

    if (file != NULL) {

        // Write formatted output to a file
        fprintf(file, "Number is %d.\n", n);
        fclose(file);
        printf("Data written to output.txt\n");
    } else {
        printf("Error opening file.\n");
    }

    return 0;
```

```
}
```

Output

Data written to output.txt

Explanation: The **fprintf** function is used to write the formatted string to the file **output.txt**. If the file is successfully opened, the formatted output is written, the file is closed, and the message "**Data written to output.txt**" is printed to the console.