

Unit 1 (Introduction to Computer)

Characteristics of Computers

- **Speed:** Computers process data and perform calculations at extremely high speeds.
 - **Accuracy:** They execute instructions with high precision, minimizing errors.
 - **Automation:** Once programmed, they can perform tasks automatically without human intervention.
 - **Storage:** Large capacity to store vast amounts of data and information.
 - **Versatility:** Capable of performing a wide range of tasks from simple calculations to complex simulations.
 - **Diligence:** Ability to perform repetitive tasks without fatigue or loss of performance.
 - **Connectivity:** Can connect and communicate with other computers and devices through networks.
 - **Multitasking:** Ability to run multiple applications and processes simultaneously.
 - **Consistency:** Deliver consistent results for the same set of inputs and instructions.
 - **Programmability:** Can be programmed to perform specific tasks and solve problems.
-

uses of computers

- **Education:** Online learning, research, virtual classrooms, and educational software.
- **Business:** Data management, financial analysis, communication, and automation of tasks.
- **Healthcare:** Patient records management, diagnostic tools, research, and telemedicine.
- **Entertainment:** Streaming media, gaming, graphic design, and content creation.
- **Communication:** Emails, video conferencing, social media, and instant messaging.
- **Science and Research:** Simulations, data analysis, experiments, and modeling.
- **Banking and Finance:** Online banking, trading, financial planning, and transactions.
- **Engineering:** Computer-aided design (CAD), simulations, and project management.
- **Government:** Public administration, record-keeping, data analysis, and e-governance.

- **Transportation:** Traffic management, logistics, vehicle design, and navigation systems.
-

Types of Computers

Computers come in various types, each designed for specific tasks and uses. Here's an overview of the main types of computers:

1. **Desktop Computers:** These are designed to be used at a single location. They offer high performance and are easily upgradeable.
 2. **Laptops:** Portable computers that integrate all the components of a desktop computer, including a display, keyboard, and touchpad.
 3. **Notebooks and Ultrabooks:** A smaller, lighter, and often more power-efficient version of laptops.
 4. **Supercomputers:** Extremely powerful computers used for complex scientific calculations, simulations, and research. They are used in fields such as climate research, molecular modeling, and physics simulations.
 5. **Smartphones:** Portable devices that combine computing capabilities with mobile phone functions.
-

Generations of Computers

1. Computers have evolved through five generations.

2. The **first generation (1940-1956)** used vacuum tubes, making them large and energy-intensive.
 3. The **second generation (1956-1963)** introduced transistors, making computers smaller, faster, and more reliable.
 4. The **third generation (1964-1971)** saw the use of integrated circuits, which further reduced size and cost while increasing speed and efficiency.
 5. The **fourth generation (1971-present)** is characterized by microprocessors, leading to personal computers and significant advancements in networking and graphical interfaces.
 6. The **fifth generation (present and beyond)** focuses on artificial intelligence, advanced parallel processing, and emerging technologies like quantum computing and the Internet of Things, aiming for enhanced human-computer interaction and problem-solving capabilities.
 7. Each generation has progressively built on technological advancements, resulting in the sophisticated systems we use today.
-

CPU

- 1:** CPU is short for Central Processing Unit. It is also known as a processor or microprocessor.
- 2:** It's one of the most important pieces of hardware in any digital computing system – if not the most important.
- 3:** Inside a CPU there are thousands of microscopic transistors, which are tiny switches that control the flow of electricity through the integrated circuits.
- 4:** CPU is located on a computer's motherboard.

5: A computer's motherboard is the main circuit board inside a computer. Its job is to connect all hardware components together.

6: Often referred to as the brain and heart of all digital systems, a CPU is responsible for doing all the work. It performs every single action a computer does and executes programs.

7: The CPU can handle interrupts, which are signals that require immediate attention. This allows the CPU to stop its current task and execute a more urgent task.

8: The CPU can perform mathematical operations like addition, subtraction, multiplication, and division, as well as logical operations like AND, OR, NOT, and XOR.

9: Modern CPUs can handle multiple threads or tasks at the same time, improving the efficiency and performance of processing.

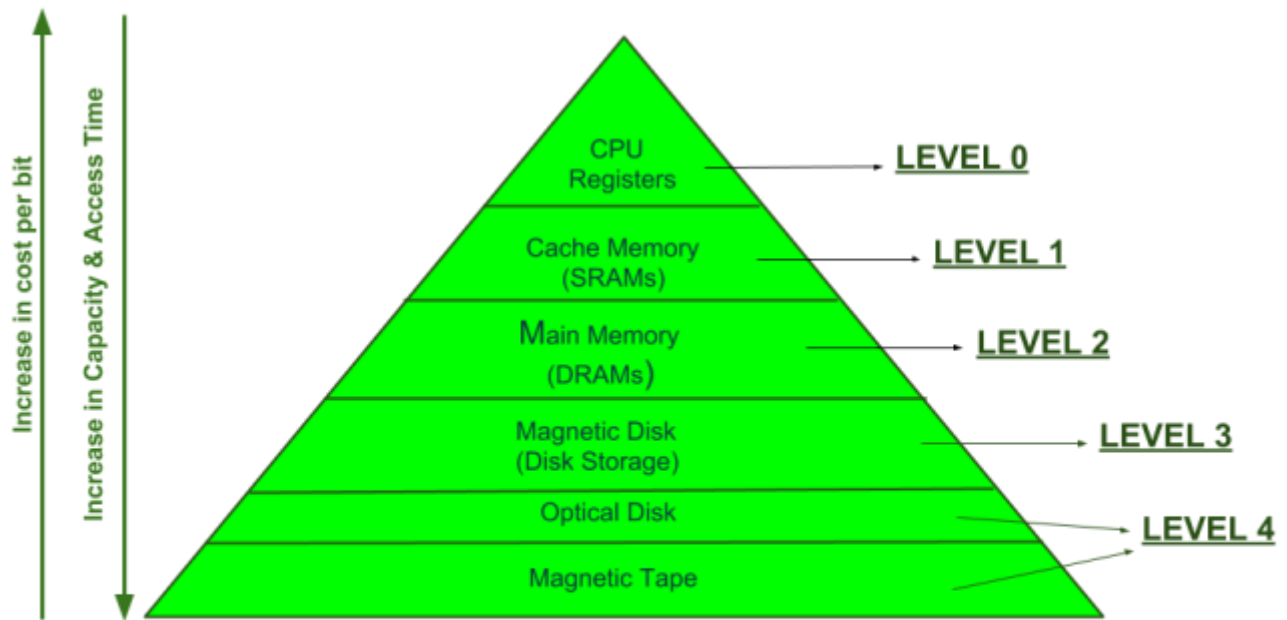
ALU

1. The Arithmetic Logic Unit (ALU) is a vital component of a computer's CPU, responsible for performing all arithmetic and logical operations.
2. It handles basic arithmetic operations like addition, subtraction, multiplication, and division, as well as logical operations such as AND, OR, NOT, and XOR.
3. The ALU also performs bitwise operations, like shifting and rotating bits, and comparison operations to determine equality and relational conditions between values.
4. The ALU consists of input operands, an opcode specifying the operation, a result register for storing outputs, and status flags indicating operation outcomes.

5. It processes data from the computer's memory and significantly impacts the CPU's overall performance and speed.
 6. By executing fundamental computations and decision-making processes, the ALU is crucial for data processing and executing complex instructions, forming the core of a computer's processing capabilities.
-

Memory hierarchy

1. The memory hierarchy in computers organizes various types of memory based on speed, cost, and capacity.
2. At the top is **registers**, located within the CPU, offering the fastest access but limited storage.
3. Below this is **cache memory**, which is slightly slower but larger, used to temporarily hold frequently accessed data for quick retrieval.
4. **Main memory (RAM)** sits mid-level, providing larger capacity but slower speed compared to cache.
5. It stores data currently in use by the CPU.
6. Below RAM is **secondary storage**, such as solid-state drives (SSDs) and hard disk drives (HDDs), which offer significant capacity at a slower speed and are used for long-term data storage.
7. At the base of the hierarchy is **tertiary storage** (e.g., magnetic tapes, optical disks), characterized by large capacity, low cost, and slow access speed, typically used for backups and archival purposes.
8. This hierarchical structure optimizes performance and cost-efficiency, ensuring the CPU has quick access to the most critical data.



MEMORY HIERARCHY DESIGN

Registers

Registers are small, high-speed storage locations within the CPU that temporarily hold data and instructions currently being processed. They are the fastest type of memory in the memory hierarchy, facilitating quick access for the CPU during computation. Key types of registers include:

- **Accumulator:** Holds intermediate arithmetic and logic results.
- **Program Counter (PC):** Contains the address of the next instruction to be executed.
- **Instruction Register (IR):** Stores the current instruction being executed.

- **General Purpose Registers:** Used for a wide range of functions as needed by the CPU.

Registers play a crucial role in speeding up the processing cycle by minimizing the time the CPU spends accessing data from slower memory types, such as cache or RAM.

Input/output (I/O) devices

Input/output (I/O) devices are peripheral components of a computer system that enable communication between the computer and the external world. They facilitate the input of data into the computer and the output of processed information to the user or other devices. Here's an overview of common I/O devices:

Input Devices:

1. **Keyboard:** Used for typing text and commands into the computer.
2. **Mouse/Trackpad:** Allows users to interact with graphical interfaces by moving a cursor.
3. **Touchscreen:** Enables touch-based input, common in smartphones, tablets, and some computers.
4. **Scanner:** Converts physical documents or images into digital format.
5. **Microphone:** Captures audio input, often used for voice commands or recording.
6. **Webcam:** Records video input, commonly used for video conferencing and streaming.
7. **Joystick/Gamepad:** Used for gaming and controlling software applications.
8. **Sensors:** Various sensors like temperature sensors, motion sensors, and biometric sensors provide environmental or user-specific input.

Output Devices:

1. **Monitor/Display:** Presents visual output from the computer, including text, graphics, and videos.
2. **Printer:** Produces hard copies of digital documents.
3. **Speakers/Headphones:** Output audio generated by the computer, including music, system sounds, and voice output.
4. **Projector:** Displays computer output onto a larger screen or surface.
5. **Plotter:** Produces high-quality graphical output, commonly used in engineering and design applications.
6. **Haptic Devices:** Provide tactile feedback to users, enhancing interaction in virtual environments or simulations.

Unit 2 (Techniques of Problem Solving)

Concept of problem solving in computers

1. Understanding the Problem:

- **Problem Definition:** Clearly define the problem you need to solve. This involves understanding the requirements, constraints, and desired outcomes.
- **Analysis:** Break down the problem into smaller, more manageable parts. Identify the inputs, outputs, and the process required to transform inputs into outputs.

2. Designing the Solution:

- **Algorithm Design:** Develop a step-by-step procedure or algorithm to solve the problem. This can involve:
- **Flowcharts:** Visual representations of the steps in an algorithm.
- **Data Structures:** Choose appropriate data structures to efficiently store and manipulate data.

3. Implementing the Solution:

- **Coding:** Translate the algorithm into a programming language. Write the code that will execute the designed solution.

4. Testing the Solution:

- **Unit Testing:** Test individual components or modules of the program to ensure they work correctly in isolation.
- **Integration Testing:** Test the combined components to ensure they work together as expected.

Key Components of Flowcharts

1. **Symbols:** Flowcharts use a set of standardized symbols to represent different types of actions and elements in a process. Common symbols include:

- Oval: Represents the start and end points of a process.
- Rectangle: Represents a process or operation step.
- Diamond: Represents a decision point that splits the flow into different branches.
- Parallelogram: Represents input or output operations.
- Arrow: Indicates the flow direction from one step to another.
- Circle: Used as a connector to link different parts of the flowchart, often used in complex flowcharts.
- Flow Lines: Arrows that connect the symbols and indicate the flow of the process from one step to the next.

2. Annotations: Additional notes or comments that provide more context or details about specific steps in the process.

3. Types of Flowcharts

- Process Flowcharts: Visualize the steps involved in a business or operational process.
- System Flowcharts: Depict the flow of data and the interactions between different components in a computer system.
- Program Flowcharts: Illustrate the sequence of operations in a computer program or algorithm.
- Data Flow Diagrams (DFDs): Show the flow of data within a system and the processing steps involved.

4. Steps to Create a Flowchart

1. Define the Process: Clearly define the process or algorithm that needs to be visualized.
2. Identify the Steps: Break down the process into individual steps or actions.
3. Determine the Flow: Decide the sequence of the steps and how they are connected.
4. Select Symbols: Choose appropriate symbols to represent each step in the process.
5. Draw the Flowchart: Use a flowcharting tool or software to create the flowchart, arranging the symbols and connecting them with flow lines.

6. Review and Refine: Check the flowchart for accuracy, completeness, and clarity. Make necessary adjustments to improve readability and correctness.
-

decision table

1. A decision table is a tabular method for representing and analyzing decision-making processes, where the various conditions and corresponding actions are listed in a systematic way.
 2. It is particularly useful for handling complex business rules and ensuring that all possible scenarios are considered.
 3. Decision tables help in clarifying the logic of decisions, improving consistency, and providing a clear documentation method.
 4. Key Components of a Decision Table
 - Conditions: These are the criteria or inputs that influence the decision. Each condition can have multiple possible values.
 - Actions: These are the outcomes or operations that result from evaluating the conditions.
 - Rules: These are combinations of conditions that define specific scenarios and determine which actions to take.
-

algorithm

1. An algorithm is a well-defined set of instructions or a step-by-step procedure used to solve a specific problem or perform a task.
2. Algorithms are fundamental to computer science and are employed in various applications ranging from simple calculations to complex data processing and artificial intelligence.
3. An algorithm must always terminate after a finite number of steps.

4. Each step of the algorithm must be precisely defined; there should be no ambiguity.
 5. An algorithm has zero or more inputs taken from a specified set of objects.
 6. An algorithm produces at least one output, which is the solution to the problem.
 7. The operations used in the algorithm are basic enough to be carried out, in principle, by a person using pencil and paper.
-

Structured programming

Structured programming is a programming paradigm aimed at improving the clarity, quality, and development time of a software program. It emphasizes the use of clear, logical structures in the code, making it easier to understand, debug, and maintain. Here are the key concepts of structured programming:

1. Sequential Structure: In a sequential structure, instructions are executed one after another in the order they are written. This is the most basic form of control flow in a program.

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

```
int main() {  
    printf("Step 1\n");  
    printf("Step 2\n");  
    printf("Step 3\n");  
    return 0;  
}
```

2. Selection Structure: Selection structures allow the program to choose different paths of execution based on conditions. The primary selection structures are `if`, `if-else`, and `switch` (or `case` in some languages).

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

```

int main() {
    int x = 10;
    if (x > 0) {
        printf("x is positive\n");
    } else {
        printf("x is non-positive\n");
    }
    return 0;
}

```

3. Repetition Structure: Repetition structures allow the program to execute a block of code multiple times. The main repetition structures are `for`, `while`, and `do-while` loops.

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

```

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

```

4. Modularity: Modularity involves breaking down a program into smaller, self-contained modules or functions. Each module performs a specific task and can be developed, tested, and debugged independently. This enhances code readability, reusability, and maintainability.

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

```

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

int main() {
    int result = add(5, 3);
    printf("Result: %d\n", result);
    return 0;
}

```

5. Encapsulation: Encapsulation involves bundling the data (variables) and methods (functions) that operate on the data into a single unit, often a class. This concept is more prominent in object-oriented programming but is also a fundamental principle

in structured programming to hide implementation details and expose only necessary components.

```
#include <stdio.h>

typedef struct {
    int width;
    int height;
} Rectangle;

int area(Rectangle rect) {
    return rect.width * rect.height;
}

int main() {
    Rectangle rect;
    rect.width = 4;
    rect.height = 5;
    printf("Area: %d\n", area(rect));
    return 0;
}
```

6. Subprograms and Functions: Subprograms (functions or procedures) are reusable blocks of code that perform a specific task. They help in reducing code redundancy and improving the organization of the program.

```
#include <stdio.h>

void greet(char *name) {
    printf("Hello, %s\n", name);
}

int main() {
    greet("Alice");
    greet("Bob");
    return 0;
}
```

Unit 3 (Planning the Computer Program)

Top-down programming

1. Top-down programming is a development method where you start with a high-level overview of a program's goal and progressively break it down into smaller, manageable tasks.
2. Begin by defining the main problem, then divide it into major sub-tasks.
3. Each sub-task is further refined into simpler tasks until they are straightforward to implement in code.
4. This approach helps in organizing and structuring code efficiently.
5. For example, to build a calculator, you would first define tasks like inputting numbers, performing operations, and displaying results, then break these down into smaller steps and implement them.
6. This method enhances clarity, modularity, and ease of debugging in software development.

Advantages:

- It provides a clear and organized structure for developing software.
- Breaking down the problem into smaller tasks promotes modular design, allowing for easier testing, debugging, and reuse of code.
- It helps in identifying and addressing potential issues early in the development process.

Disadvantages:

- Once the overall structure is defined, making significant changes to the design can be challenging and may require reworking of multiple components.

- It may be less suitable for projects where requirements are likely to change frequently, as the initial design may not easily accommodate such changes.
 - Managing the complexity of a large project with many interconnected modules can be challenging, requiring careful coordination and oversight.
-

Process of Debugging

Step 1: Reproduce the Bug

- To start, you need to recreate the conditions that caused the bug. This means making the error happen again so you can see it firsthand.
- Seeing the bug in action helps you understand the problem better and gather important details for fixing it.

Step 2: Locate the Bug

- Next, find where the bug is in your code. This involves looking closely at your code and checking any error messages or logs.
- Developers often use debugging tools to help with this step.

Step 3: Identify the Root Cause

- Now, figure out why the bug happened. Examine the logic and flow of your code and see how different parts interact under the conditions that caused the bug.
- This helps you understand what went wrong.

Step 4: Fix the Bug

- Once you know the cause, fix the code. This involves making changes and then testing the program to ensure the bug is gone.
- Sometimes, you might need to try several times, as initial fixes might not work or could create new issues.
- Using a version control system helps track changes and undo any that don't solve the problem.

Step 5: Test the Fix

After fixing the bug, run tests to ensure everything works correctly. These tests include:

- **Unit Tests:** Check the specific part of the code that was changed.
- **Integration Tests:** Verify the entire module where the bug was found.
- **System Tests:** Test the whole system to ensure overall functionality.
- **Regression Tests:** Make sure the fix didn't cause any new problems elsewhere in the application.

Step 6: Document the Process

- Finally, record what you did. Write down what caused the bug, how you fixed it, and any other important details.
- This documentation is helpful if similar issues occur in the future.

Importance of Debugging

1. Fixing mistakes in computer programming, known as bugs or errors, is necessary because programming deals with abstract ideas and concepts.
 2. Computers understand machine language, but we use programming languages to make it easier for people to talk to computers.
 3. Software has many layers of abstraction, meaning different parts must work together for an application to function properly.
 4. When errors happen, finding and fixing them can be tricky.
 5. That's where debugging tools and strategies come in handy.
 6. They help solve problems faster, making developers more efficient.
 7. This not only improves the quality of the software but also makes the experience better for the people using it.
 8. In simple terms, debugging is important because it makes sure the software works well and people have a good time using it.
-

Bottom-up programming

1. Bottom-up programming is a programming style that starts with the development of individual components and then works up to the overall structure of the program.
2. It's the opposite of top-down programming, which starts with the general concept and breaks it down into its component parts.
3. Details of each module are hidden, exposing only necessary interfaces for interaction.
4. This abstraction allows developers to work on individual components without worrying about the entire system.
5. Once individual modules are developed and tested, they are integrated to form larger subsystems.
6. These subsystems are then combined to create the complete application.
7. Testing is simpler and more manageable since each module can be tested independently.
8. Bugs are easier to isolate and fix within individual modules.

Difference between Bottom-Up Model and Top-Down Model

S. No.	TOP DOWN APPROACH	BOTTOM UP APPROACH
1.	In this approach We focus on breaking up the problem into smaller parts.	In bottom up approach we start with small parts and integrate it as whole.
2.	Mainly used by structured programming language such as COBOL, Fortran, C, etc.	Mainly used by object oriented programming language such as Java, C++, etc.
3.	Each part is programmed separately therefore contain redundancy.	Redundancy is minimized as each module is designed to perform a specific function and reuse of modules is encouraged.
4.	In this the communications is less among modules.	In this module reusability is high.
5.	It is used in debugging, module documentation, etc.	It is basic for testing and debugging.
6.	In top down approach, decomposition takes place.	In bottom up approach, integration takes place.
7.	In this top function of system might be hard to identify.	In this sometimes we start with small parts and integrate them as a whole.
8.	In this implementation details may differ.	This is not natural for the programmer.

Types of errors

In programming, errors can be broadly classified into several types. Each type of error represents a different kind of problem that can occur during the development and execution of a program. Here are the main types of programming errors:

1. Syntax Errors

- **Description:** Occur when the code violates the grammatical rules of the programming language.
- **Examples:**
 - Missing semicolons in C-like languages.
 - Incorrect indentation in Python.
 - Misspelled keywords or incorrect usage of operators.

2. Runtime Errors

- **Description:** Occur during the execution of the program, causing it to terminate abnormally. These errors are detected only when the program is run.
- **Examples:**
 - Division by zero.
 - Null pointer dereference.
 - Array index out of bounds.

3. Logical Errors

- **Description:** Occur when the program runs without crashing but produces incorrect results. These errors are due to flaws in the logic or algorithm.
- **Examples:**
 - Incorrect formula used in calculations.

- Incorrect loop conditions causing infinite loops or wrong iterations.
- Misplaced conditional statements.

4. Compilation Errors

- **Description:** Occur when the code cannot be compiled into an executable due to errors in the source code.
 - **Examples:**
 - Missing header files or import statements.
 - Type mismatch errors in statically typed languages.
-

Documentation in programming

Documentation is a vital part of programming that helps users prepare input, process code, and get output. It also makes it easier to maintain code. Documentation can include:

1. User documentation

Also known as user manuals or guides, this type of documentation helps users understand how to use a product and troubleshoot issues. It can include instructions, explanations, and reference materials.

2. External documentation

This type of documentation helps users understand and navigate a product or service without contacting customer support. It can help users learn new features, solve problems, or integrate systems.

3. Code documentation

This type of documentation describes what a codebase does and how it can be used. It can include textual explanations, pictures, developer handbooks, application summaries, API documentation, and more.

Unit 4 (Introduction to C)

History of C

1. C was developed by Dennis Ritchie at Bell Labs in the early 1970s as a system programming language for Unix.
 2. It evolved from the B language, which was itself based on BCPL.
 3. C provided low-level access to memory, efficient performance, and a simple syntax, making it ideal for operating system and compiler development.
 4. Unix, originally written in assembly, was rewritten in C in 1973, demonstrating C's capabilities.
 5. Over time, C became popular for its portability and efficiency, influencing many subsequent languages like C++, C#, and Java.
 6. The American National Standards Institute (ANSI) standardized C in 1989, further cementing its widespread adoption and enduring legacy in software development.
 7. C remains a dominant language in system/software development, embedded systems, and performance-critical applications, showcasing its lasting influence.
-

C Character Set

The C character set consists of:

- **Letters:** A-Z (uppercase) and a-z (lowercase)
- **Digits:** 0-9

- **Special Characters:** + - * / = % & ^ ! ~ | \ ? : . , ; ' " () [] { } < > # _
 - **Whitespace Characters:** space, tab, newline, vertical tab, form feed
-

Tokens

Tokens are the basic building blocks of a C program. They include:

- **Keywords:** Reserved words with special meaning (e.g., `int`, `return`, `if`, `else`)
 - **Identifiers:** Names given to variables, functions, and other user-defined items
 - **Constants:** Fixed values that do not change during program execution (e.g., `42`, `3.14`, `'a'`)
 - **String Literals:** Sequence of characters enclosed in double quotes (e.g., `"Hello, World!"`)
 - **Operators:** Symbols that specify operations (e.g., `+`, `-`, `*`, `/`)
 - **Punctuators:** Symbols that have special meaning in C syntax (e.g., `{`, `}`, `;`, `,`)
-

Constants

Constants are immutable values used in programs. Types include:

- **Integer Constants:** Whole numbers (e.g., `100`, `-45`)
 - **Floating-point Constants:** Numbers with fractional parts (e.g., `3.14`, `-0.001`)
 - **Character Constants:** Single characters enclosed in single quotes (e.g., `'a'`, `'\n'`)
 - **String Constants:** Sequence of characters enclosed in double quotes (e.g., `"hello"`)
 - **Symbolic Constants:** Named constants defined using the `#define` preprocessor directive (e.g., `#define PI 3.14`)
-

Variables

Variables are named storage locations used to hold data that can be modified during program execution. They must be declared before use, specifying the variable's type and optionally initializing it (e.g., `int x = 10;`).

Keywords

Keywords are reserved words with predefined meanings and cannot be used as identifiers. Examples include:

- `auto`
 - `break`
 - `case`
 - `char`
 - `const`
 - `continue`
 - `default`
 - `do`
 - `double`
-

Identifiers

Identifiers are names given to various program elements such as variables, functions, arrays, and structures. They must begin with a letter or an underscore followed by letters, digits, or underscores (e.g., `myVariable`, `_temp`, `compute_sum`). Identifiers are case-sensitive and must not conflict with keywords.

Types of Operators

Arithmetic Operators

Used for mathematical operations:

- $+$ (Addition): $a + b$
- $-$ (Subtraction): $a - b$
- $*$ (Multiplication): $a * b$
- $/$ (Division): a / b
- $\%$ (Modulus): $a \% b$

Relational Operators

Used to compare two values:

- $==$ (Equal to): $a == b$
- $!=$ (Not equal to): $a != b$
- $>$ (Greater than): $a > b$
- $<$ (Less than): $a < b$
- $>=$ (Greater than or equal to): $a >= b$
- $<=$ (Less than or equal to): $a <= b$

Logical Operators

Used to combine multiple conditions:

- $\&\&$ (Logical AND): $a \&\& b$
- $\|\|$ (Logical OR): $a \|\| b$
- $!$ (Logical NOT): $!a$

Bitwise Operators

Used for bit-level operations:

- $\&$ (Bitwise AND): $a \& b$
- $|$ (Bitwise OR): $a | b$
- \wedge (Bitwise XOR): $a \wedge b$
- \sim (Bitwise NOT): $\sim a$
- \ll (Left shift): $a \ll 2$
- \gg (Right shift): $a \gg 2$

Assignment Operators

Used to assign values to variables:

- `=` (Assignment): `a = b`
- `+=` (Add and assign): `a += b`
- `-=` (Subtract and assign): `a -= b`
- `*=` (Multiply and assign): `a *= b`
- `/=` (Divide and assign): `a /= b`
- `%=` (Modulus and assign): `a %= b`
- `&=` (Bitwise AND and assign): `a &= b`
- `|=` (Bitwise OR and assign): `a |= b`
- `^=` (Bitwise XOR and assign): `a ^= b`
- `<<=` (Left shift and assign): `a <<= b`
- `>>=` (Right shift and assign): `a >>= b`

Increment and Decrement Operators

Used to increase or decrease the value of a variable by one:

- `++` (Increment): `a++` or `++a`
- `--` (Decrement): `a--` or `--a`

Conditional (Ternary) Operator

A shorthand for if-else statements:

- `?:` (Ternary): `condition ? expression1 : expression2`

Comma Operator

Used to separate two or more expressions:

- `,` (Comma): `a = (b = 3, b + 2)`

Type Cast Operator

Used to convert a variable from one type to another:

- `(type)` (Type casting): `(int) a`

Sizeof Operator

Used to determine the size of a data type or a variable:

- `sizeof` (Sizeof): `sizeof(a)` or `sizeof(int)`

Member Access Operators

Used to access members of structures and unions:

- `.` (Direct member access): `structVar.member`
- `->` (Pointer to member access): `structPtr->member`

Operator Precedence

The following table lists the order of precedence of operators in C. Here, operators with the highest precedence appear at the top of the table, and those with the lowest appear at the bottom.

Category	Operator	Associativity
Postfix	() [] -> . ++ --	Left to right
Unary	+ - ! ~ ++ -- (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR		Left to right
Logical AND	&&	Left to right
Logical OR		Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= =	Right to left
Comma	,	Left to right

Within an expression, higher precedence operators will be evaluated first.

Algorithm

An algorithm in C, as in any programming language, is a step-by-step procedure or a set of rules designed to perform a specific task or solve a particular problem. It is a logical sequence of actions to achieve a desired outcome. While an algorithm is a conceptual idea and not limited to any programming language, it can be implemented in C through structured code.

Characteristics of an Algorithm

1. **Definiteness:** Each step of the algorithm is clearly and unambiguously defined.
2. **Finiteness:** The algorithm must terminate after a finite number of steps.
3. **Input:** An algorithm has zero or more inputs.
4. **Output:** An algorithm has one or more outputs.
5. **Effectiveness:** Each step must be basic enough to be carried out, in principle, by a person using only pencil and paper.

Example of an Algorithm in C

Step-by-Step Algorithm

1. Start.
2. Initialize the variable `max` with the value of the first element in the array.
3. Iterate through each element of the array.
4. Compare each element with `max`.
5. If the element is greater than `max`, update `max` with this element.
6. After completing the loop, `max` will hold the largest element in the array.
7. End.

C Implementation

Here's how you can implement this algorithm in C:

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

```

int findLargest(int arr[], int n) {
    int max = arr[0]; // Step 2
    for (int i = 1; i < n; i++) { // Step 3
        if (arr[i] > max) { // Step 4
            max = arr[i]; // Step 5
        }
    }
    return max; // Step 6
}

int main() {
    int arr[] = {10, 45, 2, 33, 56, 78, 23};
    int n = sizeof(arr) / sizeof(arr[0]);

    int largest = findLargest(arr, n); // Calling the function

    printf("The largest number in the array is: %d\n", largest); //
Step 7

    return 0;
}

```

Flowchart

A flowchart is a visual representation of the sequence of steps and decisions needed to perform a process or solve a problem. It uses various symbols to depict different types of actions and the flow of control between them. Flowcharts are commonly used in planning, analyzing, designing, and documenting processes and algorithms.

Components of a Flowchart

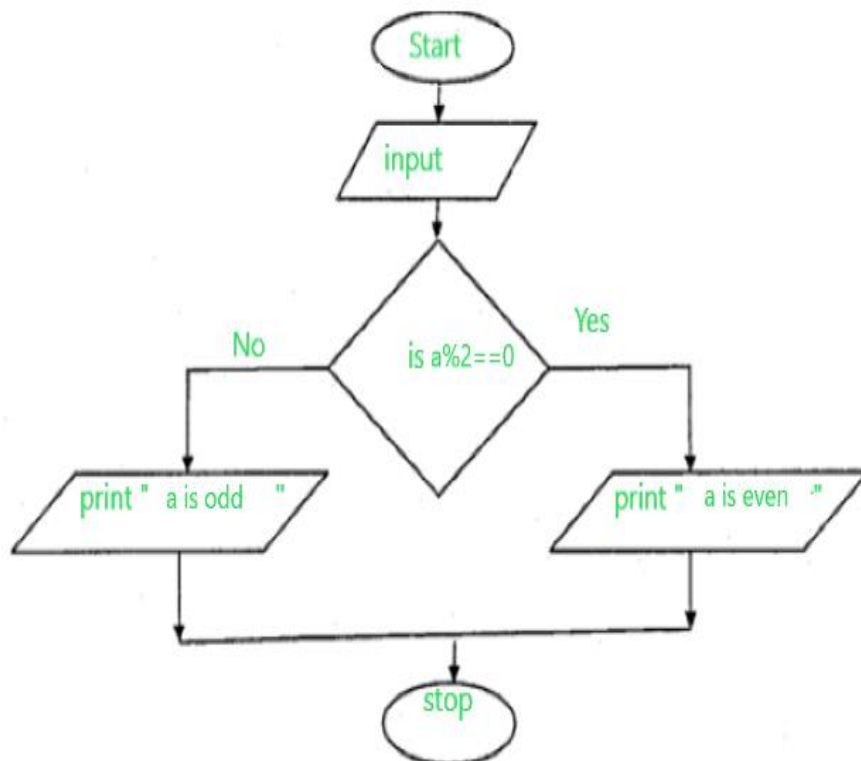
1. **Start/End (Terminator):** Represented by an oval, indicating the beginning or end of a process.
2. **Process:** Represented by a rectangle, showing a step or action in the process.
3. **Decision:** Represented by a diamond, indicating a decision point with different branches for true/false or yes/no outcomes.
4. **Input/Output:** Represented by a parallelogram, showing input to or output from a process.

5. **Arrow (Flow Line):** Indicates the direction of flow from one step to another.
6. **Connector:** Represented by a small circle, used to connect different parts of the flowchart, especially when it spans multiple pages.

Difference in flowchart and alogrithm

S. No	Algorithm	Flowchart
1.	An algorithm is a step-by-step procedure to solve a problem.	A flowchart is a diagram created with different shapes to show the flow of data.
2.	The algorithm is complex to understand.	A flowchart is easy to understand.
3.	In the algorithm, plain text is used.	In the flowchart, symbols/shapes are used.
4.	The algorithm is easy to debug.	A flowchart is hard to debug.
5.	The algorithm is difficult to construct.	A flowchart is simple to construct.
6.	The algorithm does not follow any rules.	The flowchart follows rules to be constructed.
7.	The algorithm is the pseudo-code for the program.	A flowchart is just a graphical representation of that logic.

flowchart to check whether the input number is odd or even



Unit 5 (Decision Making and Looping)

If statement

The if statement works by first evaluating the condition in parentheses. If the condition is true, the code inside the curly braces is executed. If the condition is false, the code inside the else block is executed.

```
//example
#include <stdio.h>
int main() {
    int age = 18;

    if (age >= 18) {
        printf("You are eligible to vote.\n");
    } else {
        printf("You are not eligible to vote.\n");
    }

    return 0;
}
```

if else if ladder

if else if ladder *in C programming* is used to test a series of conditions sequentially. Furthermore, if a condition is tested only when all previous if conditions in the if-else ladder are false. If any of the conditional expressions evaluate to be true, the appropriate code block will be executed, and the entire if-else ladder will be terminated.

```
// C Program to check whether
// a number is positive, negative
// or 0 using if else if ladder
#include <stdio.h>

int main()
{
    int n = 0;

    // all Positive numbers will make this
    // condition true
    if (n > 0) {
        printf("Positive");
    }

    // all Negative numbers will make this
    // condition true
    else if (n < 0) {
        printf("Negative");
    }
}
```

```
// if a number is neither Positive nor Negative
else {
printf("Zero");
}
return 0;
}
```

nested if-else statement

A nested if-else statement is an if statement inside another if statement.

the outer *if statement* has two possible paths: one for when the condition is *true*, and another for when the condition is *false*. If the condition is true, the program will execute the code inside the block associated with the outer if statement. However, if the condition is false, the program will skip over that block and move to the else block. Within the outer if block, there is another if statement, which can also have two possible paths depending on whether the condition is true or false.

```
#include <stdio.h>
int main() {
    int num;

    printf("Enter a number: ");
    scanf("%d", &num);

    if (num > 0) {
        printf("%d is positive.\n", num);
    } else {
        if (num < 0) {
            printf("%d is negative.\n", num);
        } else {
            printf("%d is zero.\n", num);
        }
    }

    return 0;
}
```

Switch case

Switch case statement evaluates a given expression and based on the evaluated value(matching a certain condition), it executes the statements associated with it. Basically, it is used to perform different actions based on different conditions(cases).

- Switch case statements follow a selection-control mechanism and allow a value to change control of execution.
- They are a substitute for long [*if statements*](#) that compare a variable to several integral values.
- The switch statement is a multiway branch statement. It provides an easy way to dispatch execution to different parts of code based on the value of the expression.

```
// C program to Demonstrate returning of day based numeric
// value
#include <stdio.h>

int main()
{
    // switch variable
    int var = 1;

    // switch statement
    switch (var) {
        case 1:
            printf("Case 1 is Matched.");
            break;

        case 2:
            printf("Case 2 is Matched.");
            break;

        case 3:
            printf("Case 3 is Matched.");
            break;

        default:
            printf("Default case is Matched.");
            break;
    }

    return 0;
}
```

break statement

The break statement ends the loop immediately when it is encountered.

```
// Program to calculate the sum of numbers (10 numbers max)
// If the user enters a negative number, the loop terminates

#include <stdio.h>

int main() {
    int i;
    double number, sum = 0.0;

    for (i = 1; i <= 10; ++i) {
        printf("Enter n%d: ", i);
        scanf("%lf", &number);

        // if the user enters a negative number, break the loop
        if (number < 0.0) {
            break;
        }

        sum += number; // sum = sum + number;
    }

    printf("Sum = %.2lf", sum);

    return 0;
}
```

continue statement

The continue statement skips the current iteration of the loop and continues with the next iteration.

```
// Program to calculate the sum of numbers (10 numbers max)
// If the user enters a negative number, it's not added to the result

#include <stdio.h>
int main() {
    int i;
    double number, sum = 0.0;
```

```
for (i = 1; i <= 10; ++i) {
    printf("Enter a n%d: ", i);
    scanf("%lf", &number);

    if (number < 0.0) {
        continue;
    }

    sum += number; // sum = sum + number;
}

printf("Sum = %.2lf", sum);

return 0;
}
```

while Loop Structure

The while loop works by following a very structured top-down approach that can be divided into the following parts:

1. **Initialization:** In this step, 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
2. **Conditional Statement:** This is one of the most crucial steps 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**.
3. **Body:** It is the actual set of statements that will be executed till the specified condition is true. It is generally enclosed inside **{ } braces**.
4. **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.

```
// C program to demonstrate while loop
#include <stdio.h>

int main()
{
    // Initialization of loop variable
    int i = 0;

    // setting test expression as (i < 5), means the loop
    // will execute till i is less than 5
    while (i < 5) {

        // loop statements
        printf("GeeksforGeeks\n");

        // updating the loop variable
        i++;
    }
    return 0;
}
```

do while loop

The **do...while in C** is a loop statement used to repeat some part of the code till the given condition is fulfilled. It is a form of an **exit-controlled or post-tested loop** where the test condition is checked after executing the body of the loop. Due to this, the statements in the do...while loop will always be executed at least once no matter what the condition is.

```
// C Program to demonstrate the use of do...while loop
#include <stdio.h>

int main()
{

    // loop variable declaration and initialization
    int i = 0;
    // do while loop
    do {
        printf("Geeks\n");
        i++;
    } while (i < 3);
}
```



```
return 0;
}
```

Structure of for Loop

The for loop follows a very structured approach where it begins with initializing a condition then checks the condition and in the end executes conditional statements followed by an updation of values.

1. **Initialization:** This step initializes a loop control variable with an initial value that helps to progress the loop or helps in checking the condition. It acts as the index value when iterating an array or string.
2. **Check/Test Condition:** This step of the **for loop** defines the condition that determines whether the loop should continue executing or not. The condition is checked before each iteration and if it is true then the iteration of the loop continues otherwise the loop is terminated.
3. **Body:** It is the set of statements i.e. variables, functions, etc that is executed repeatedly till the condition is true. It is enclosed within curly braces { }.
4. **Updation:** This specifies how the loop control variable should be updated after each iteration of the loop. Generally, it is the incrementation (variable++) or decrementation (variable--) of the loop control variable.

```
// C program to demonstrate for loop
#include <stdio.h>

int main()
{
    int gfg = 0;

    for (gfg = 1; gfg <= 5; gfg++)
    {

        printf("GeeksforGeeks\n");
    }
}
```

```
}  
return 0;  
}
```

Unit 6 (Arrays and Strings)

Array

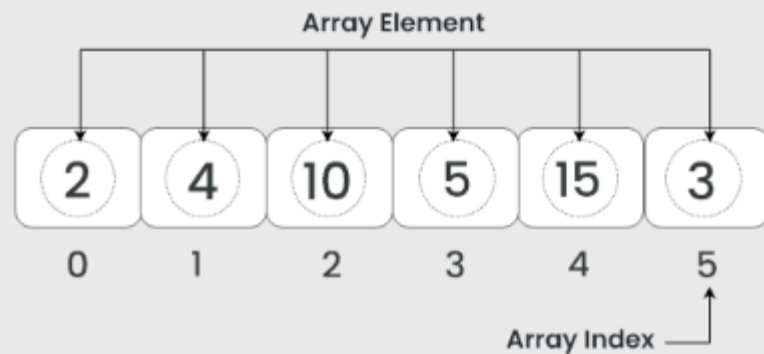
1. Array is a linear data structure where all elements are arranged sequentially.
2. It is a collection of elements of same data type stored at contiguous memory locations.

3. In C language, the array has a fixed size meaning once the size is given to it, it cannot be changed.
4. It's one of the most popular and simple data structures and is often used to implement other data structures.
5. Each item in an array is indexed starting with 0.
6. Each element in an array is accessed through its index.
7. Arrays are used in a wide variety of applications like, storing data for processing, implementing data structures such as stacks and queues and representing data in tables and matrices.



Array

Data Structure



Initialization of One-Dimensional Array

```
int numbers[5] = {10, 20, 30, 40, 50}; // Initialize with specific values
```

Initialization of Two-Dimensional Array

```
int matrix[2][3] = {{1, 2, 3}, {4, 5, 6}}; // Initialize with specific values
```

Accessing elements in Array

Accessing elements in a one-dimensional array in C is straightforward. You use the array name followed by the index of the element you want to access, enclosed in square brackets. Array indices start at 0, so the first element is at index 0, the second element is at index 1, and so on.

```
#include <stdio.h>

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

    // Accessing elements
    printf("First element: %d\n", numbers[0]); // Outputs: 10
    printf("Second element: %d\n", numbers[1]); // Outputs: 20
    printf("Third element: %d\n", numbers[2]); // Outputs: 30
    printf("Fourth element: %d\n", numbers[3]); // Outputs: 40
    printf("Fifth element: %d\n", numbers[4]); // Outputs: 50

    return 0;
}
```

String

1. In C, a string is a sequence of characters.
2. Unlike some other programming languages, C does not have a built-in string data type. Instead, strings are represented as arrays of characters.

3. Strings are stored as arrays of characters.
4. The null character ('\0') marks the end of the string.
5. The C Standard Library provides a set of functions for string manipulation.
6. Since strings are essentially arrays of characters, they can be accessed and modified using array indexing.
7. However, care must be taken to ensure that the null terminator is preserved, as its absence can lead to undefined behavior when manipulating strings.

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

int main() {
    // Initializing a string
    char greeting[] = "Hello, World!";

    // Printing the string
    printf("%s\n", greeting);

    return 0;
}
```

String Functions in C

strcpy	Copies a string into another
strncpy	Copies first n characters of one string into another
strcmp	Compares two strings
strncmp	Compares first n characters of two strings
strcmpi	Compares two strings without regard to case ("i" denotes that this function ignores case)
stricmp	Compares two strings without regard to case (identical to strcmpi)
strnicmp	Compares first n characters of two strings without regard to case
strdup	Duplicates a string
strchr	Finds first occurrence of a given character in a string
strrchr	Finds last occurrence of a given character in a string
strstr	Finds first occurrence of a given string in another string
strset	Sets all characters of string to a given character
strnset	Sets first n characters of a string to a given character
strrev	Reverses string

String Functions example

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

int main() {
```

```

// Declaring strings
char str1[100] = "Hello, ";
char str2[100] = "world!";
char str3[100];

// 1. strlen: Get the length of a string
int len = strlen(str1);
printf("Length of str1: %d\n", len);

// 2. strcpy: Copy a string
strcpy(str3, str2);
printf("str3 after strcpy: %s\n", str3);

// 3. strcat: Concatenate two strings
strcat(str1, str2);
printf("str1 after strcat: %s\n", str1);

// 4. strcmp: Compare two strings
int cmp = strcmp(str1, "Hello, world!");
if (cmp == 0) {
    printf("str1 is equal to \"Hello, world!\"\n");
} else {
    printf("str1 is not equal to \"Hello, world!\"\n");
}

// 5. strstr: Find a substring in a string
char *substr = strstr(str1, "world");
if (substr) {
    printf("\"world\" found in str1 at position: %ld\n", substr -
str1);
} else {
    printf("\"world\" not found in str1\n");
}

return 0;
}

```

Unit 7 (Functions and Pointers)

Functions

1. A function in C is a block of code that performs a specific task.
 2. Functions can be called from anywhere in the program.
 3. To call a function, you can use the function name followed by the argument list in parentheses.
 4. Functions are used to modularize and organize code.
 5. They provide a way to break down a program into smaller, manageable units, which makes the code more readable, reusable, and maintainable.
 6. Since functions encapsulate specific tasks, making changes to the program becomes easier. You can modify a function's code without affecting other parts of the program, leading to better maintainability.
 7. Functions can be grouped into libraries (header files in C) and reused across multiple projects. This promotes consistency and efficiency in development.
-

Need of Functions

1. Functions help to organize code by grouping related statements together. This makes the code more structured and easier to navigate.
2. Functions prevent code duplication by allowing the same block of code to be used multiple times across a program. This reduces redundancy and makes the codebase smaller and more efficient.
3. By breaking down a program into smaller, independent functions, each function can focus on a specific task. This modularity makes it easier to understand, develop, test, and maintain the code.
4. Functions can be reused across different programs or in different parts of the same program.
5. Functions allow programmers to abstract complex operations into simple function calls.
6. Isolating code into functions makes it easier to test and debug specific parts of the program.

7. Functions can take parameters, allowing them to handle different inputs and perform a wide range of tasks.
-

scope of variables

The scope of a variable refers to the region of the program where the variable is accessible. There are different types of scope:

1. **Block Scope:**
 - Variables declared within a block (enclosed by `{ }`) have block scope.
 - These variables are only accessible within the block where they are declared.
 - Example: Variables declared inside a function or within a `for` loop.
 2. **Function Scope:**
 - Labels (used with `goto` statements) have function scope.
 - They are visible throughout the function in which they are declared.
 3. **File Scope:**
 - Variables declared outside of any function (at the top level of a file) have file scope.
 - These variables are accessible from the point of declaration to the end of the file.
 4. **Global Scope:**
 - Global variables are those declared outside of all functions.
 - They are accessible from any function within the same file or other files if the variable is declared using the `extern` keyword.
-

Lifetime of Variables

The lifetime of a variable refers to the duration for which the variable exists in memory during program execution. Different storage classes determine the lifetime:

1. **Automatic (Local) Variables:**

- These variables are declared within a block or function.
- They have automatic storage duration, meaning they are created when the block or function is entered and destroyed when it is exited.
- The default storage class for local variables is `auto`.

2. **Static Variables:**

- These variables retain their value between function calls.
- They are initialized only once and exist for the entire duration of the program.
- Static variables can be declared within a function or at the file level.
- If declared within a function, their scope is limited to that function, but their lifetime is the entire program execution.
- If declared at the file level, they are accessible only within that file (file scope).

3. **Global Variables:**

- Global variables have static storage duration, meaning they exist for the entire duration of the program.
- They are initialized at the start of the program and destroyed at its termination.

4. **Dynamic Variables:**

- These variables are allocated and deallocated manually using functions like `malloc`, `calloc`, `realloc`, and `free`.
- They exist until explicitly deallocated or the program ends.

Function calls

A function call is the mechanism by which a program transfers control to a function. When a function is called, the program execution transfers to the function's definition, executes its statements, and then returns control to the point immediately after the function call.

1. Call by Value:

Call by Value is a method of passing arguments to a function where the actual value (a copy) of the variable is passed to the function parameters. This means:

- The function receives a copy of the argument's value.
- Changes made to the parameter inside the function do not affect the original variable outside the function.
- In C, all arguments are passed by value by default.

```
#include <stdio.h>

void increment(int x) {
    x++;
    printf("Inside function: %d\n", x); // Prints incremented value
}

int main() {
    int num = 5;
    increment(num);
    printf("Outside function: %d\n", num); // Prints original value
    return 0;
}
```

2. Call by Reference:

Call by Reference is a method of passing arguments to a function where the address (reference) of the variable is passed instead of the actual value. This allows the function to directly access and modify the original variable's value.

- In C, call by reference is achieved using pointers.
- Changes made to the parameter inside the function affect the original variable outside the function.

```
#include <stdio.h>

void increment(int *x) {
    (*x)++;
    printf("Inside function: %d\n", *x); // Prints incremented value
}

int main() {
    int num = 5;
    increment(&num); // Passing address of num
    printf("Outside function: %d\n", num); // Prints modified value
    return 0;
}
```

Return Values

Functions in C can return values using the `return` statement. The return type of the function must be specified in the function declaration and definition. A function can return a value of any data type, including `int`, `float`, `char`, or even `void` if it doesn't return any value.

```
#include <stdio.h>

int add(int a, int b) {
    return a + b; // Returns the sum of a and b
}

int main() {
    int result = add(3, 4);
    printf("Result: %d\n", result); // Prints 7
    return 0;
}
```

category of functions

1. No Argument, No Return Value

Functions in this category do not take any arguments (parameters) and do not return any value (void functions).

Characteristics :

- Defined with `void` return type (`void functionName() { ... }`).
- Does not accept any parameters.
- Performs actions or operations but does not return any result.

```
#include <stdio.h>

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

int main() {
    greet(); // Function call with no arguments and no return value
    return 0;
}
```

2. No Argument, With Return Value

Functions in this category do not take any arguments (parameters) but return a value.

Characteristics :

- - Defined with a return type other than `void` (`int`, `float`, `char`, etc.).
 - Does not accept any parameters.
 - Performs computations or operations and returns a result of the specified type.

```
#include <stdio.h>

int getRandomNumber() {
    return rand(); // Return a random number
}

int main() {
    int num = getRandomNumber(); // Function call with no arguments
    // but returns a value
    printf("Random number: %d\n", num);
    return 0;
}
```

3. With Argument, With Return Value

Functions in this category take one or more arguments (parameters) and return a value.

Characteristics :

- Defined with a return type other than `void`.

- Accepts one or more parameters.
- Performs computations or operations using the provided arguments and returns a result.

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

```
int add(int a, int b) {  
    return a + b; // Return the sum of a and b  
}
```

```
int main() {  
    int result = add(3, 4); // Function call with arguments and  
    returns a value  
    printf("Sum: %d\n", result);  
    return 0;  
}
```

Recursion

1. Recursion is a programming technique where a function calls itself to solve smaller instances of the same problem.
2. This self-referential approach can break down complex problems into more manageable sub-problems, making the solution easier to conceptualize and implement.
3. Recursion simplifies complex problems by breaking them down into smaller, more manageable sub-problems that are easier to solve.
4. Recursion often follows a divide-and-conquer approach, where a problem is divided into smaller sub-problems, each solved recursively.
5. Recursive functions use the call stack to keep track of function calls.
6. Each recursive call adds a new frame to the stack, and when a base case is reached, the stack unwinds as the results are combined.
7. Without a proper base case, recursion can lead to infinite loops and stack overflow errors.
8. Ensuring a well-defined base case is crucial for preventing such issues.

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

```
// Function to calculate factorial using recursion
int factorial(int n) {
    // Base case: factorial of 0 or 1 is 1
    if (n == 0 || n == 1) {
        return 1;
    }
    // Recursive case: n! = n * (n-1)!
    else {
        return n * factorial(n - 1);
    }
}

int main() {
    int number;

    // Prompt user for input
    printf("Enter a positive integer: ");
    scanf("%d", &number);

    // Check if the input is non-negative
    if (number < 0) {
        printf("Factorial is not defined for negative numbers.\n");
    } else {
        // Calculate and print the factorial
        printf("Factorial of %d is %d\n", number, factorial(number));
    }

    return 0;
}
```

Pointers

1. Pointers are one of the core components of the C programming language.
2. A pointer can be used to store the memory address of other variables, functions, or even other pointers.
3. The use of pointers allows low-level memory access, dynamic memory allocation, and many other functionality in C.

4. Pointers allow dynamic allocation of memory at runtime, this enables the creation of data structures whose size can be determined during the execution of the program, such as dynamic arrays, linked lists, and trees.
 5. Pointers provide a way to directly access and manipulate memory addresses.
 6. Pointers allow passing large structures or arrays to functions without copying the entire data.
 7. Pointers are essential for implementing complex data structures such as linked lists, trees, graphs, and hash tables.
 8. Pointers provide a high degree of flexibility in programming, allowing complex data manipulations and efficient implementation of algorithms.
 9. They enable programmers to create and manage complex data relationships and structures.
-

Declaration of Pointers

To declare a pointer, you specify the type of data it points to, followed by an asterisk (*), and then the pointer's name.

```
int *ptr;    // Declares a pointer to an integer
char *ch;    // Declares a pointer to a character
float *fptr; // Declares a pointer to a float
```

Initialization of Pointers

Pointers are typically initialized to the address of an existing variable using the address-of operator (&).

```
int var = 10;
int *ptr = &var; // ptr now holds the address of var
```

Dereferencing a Pointer

Dereferencing a pointer means accessing the value stored at the memory address the pointer holds. This is done using the asterisk (*) operator.

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

printf("%d", *ptr); // Outputs 10, the value of var
```

Pointer Arithmetic

Pointers can be incremented and decremented, which allows for traversal through arrays and other data structures. Pointer arithmetic is done based on the size of the data type the pointer points to.

```
int arr[] = {10, 20, 30};
int *ptr = arr;

printf("%d", *ptr); // Outputs 10
ptr++;
printf("%d", *ptr); // Outputs 20
```

Null Pointers

A null pointer is a pointer that does not point to any valid memory location. It is typically used to signify that the pointer is not intended to point to any object.

```
int *ptr = NULL;
```

Pointers to Pointers

C allows the creation of pointers to pointers, which can be used to create complex data structures like linked lists, trees, etc.

```
int var = 10;
int *ptr = &var;
int **pptr = &ptr;

printf("%d", **pptr); // Outputs 10
```

Function Pointers

Pointers can also be used to point to functions, allowing for dynamic function calls and callback mechanisms.

```
void func() {
    printf("Hello, World!");
}

void (*fptr)() = func;
fptr(); // Calls func and prints "Hello, World!"
```

Unit 8 (Structures and Unions)

Structures

1. Structures in C are a user-defined data type that allows the combination of data items of different kinds.
2. Structures are used to represent a record, grouping related variables under one name.
3. Each element in a structure is called a member, and these members can be of different types.
4. A structure is defined with the struct keyword followed by the structure name and a block of members enclosed in curly braces.
5. The definition specifies the layout of the data but does not allocate memory.
6. Members are variables that can be of various data types, including basic types (int, float, char), arrays, and even other structures.
7. Each member within the structure has a unique name.
8. After defining a structure, you can create instances (or variables) of that structure type.
9. Each instance of the structure has its own copy of the members.
10. Members of a structure instance are accessed using the dot operator (.). The syntax is 'instanceName.memberName.'

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

```

// Define a structure to represent a person
struct Person {
    char name[50];
    int age;
    float height;
};

int main() {
    // Declare a structure variable
    struct Person person1;

    // Initialize the structure members
    printf("Enter name: ");
    scanf("%s", person1.name);

    printf("Enter age: ");
    scanf("%d", &person1.age);

    printf("Enter height: ");
    scanf("%f", &person1.height);

    // Access and display the structure members
    printf("\nPerson Details:\n");
    printf("Name: %s\n", person1.name);
    printf("Age: %d\n", person1.age);
    printf("Height: %.2f\n", person1.height);

    return 0;
}

```

Arrays of structures

Arrays of structures allow you to create an array where each element is a structure. This is useful for handling collections of records.

```

#include <stdio.h>

// Define a structure to represent a person
struct Person {
    char name[50];

```

```

    int age;
    float height;
};

int main() {
    // Declare an array of structures
    struct Person people[3];

    // Initialize the structure members for each person in the array
    for (int i = 0; i < 3; i++) {
        printf("Enter details for person %d:\n", i + 1);

        printf("Enter name: ");
        scanf("%s", people[i].name);

        printf("Enter age: ");
        scanf("%d", &people[i].age);

        printf("Enter height: ");
        scanf("%f", &people[i].height);


        printf("\n");
    }

    // Display the details of each person in the array
    printf("Details of people:\n");
    for (int i = 0; i < 3; i++) {
        printf("Person %d:\n", i + 1);
        printf("Name: %s\n", people[i].name);
        printf("Age: %d\n", people[i].age);
        printf("Height: %.2f\n", people[i].height);
        printf("\n");
    }

    return 0;
}

```

Difference between Array and Structure

Aspect	Array	Structure
Definition	Collection of elements of the same data type	Collection of elements (members) of different data types
Memory Allocation	Allocated in contiguous memory locations	Members are allocated in contiguous memory locations but may have padding between them
Accessing Elements	Accessed using indices (e.g., <code>arr[0]</code> , <code>arr[1]</code>)	Accessed using dot operator (e.g., <code>structVar.member</code>)
Data Types	Must be of the same data type	Can be of different data types
Syntax	<code>dataType arrayName[arraySize];</code>	<code>struct structName { memberDefinitions }; struct structName varName;</code>
Homogeneity	Homogeneous (all elements are of the same type)	Heterogeneous (members can be of different types)
Usage	Suitable for storing multiple values of the same type	Suitable for grouping related data of different types
Iteration	Can be easily iterated over using loops	Requires individual access to each member
Example	<code>int arr[5];</code>	<code>struct Person { char name[50]; int age; float height; };</code>
Dynamic Memory	Can be dynamically allocated using pointers	Can be dynamically allocated, but each instance has a fixed layout
Operations	Supports standard operations like sorting, searching 	Operations depend on the types of individual members

union

1. A union is a data structure that allows storing different data types in the same memory location.
 2. Like structures, unions enable grouping of variables under a single name.
 3. Unlike structures, which allocate enough memory for each member separately, unions allocate memory large enough to hold the largest member. However, only one member can occupy the union's memory at a time.
 4. All members of a union share the same memory space, unlike structures where each member has its allocated memory space.
 5. By using a union, the system can minimize memory usage, which is critical in resource-constrained environments where every byte counts.
 6. Additionally, unions provide flexibility in accessing and interpreting the same memory location in different ways, depending on the context or application needs.
-

Union Example

```
#include <stdio.h>

// Define a union to store different data types in the same memory
location
union Data {
    int intValue;
    float floatValue;
    char stringValue[20];
};

int main() {
    union Data data;

    // Setting values in the union
    data.intValue = 10;
    printf("intValue: %d\n", data.intValue);

    data.floatValue = 3.14;
    printf("floatValue: %.2f\n", data.floatValue);
}
```

```

strcpy(data.stringValue, "Hello, Union!");
printf("stringValue: %s\n", data.stringValue);

// Accessing the same memory location in different ways
printf("\nAccessing union members:\n");
printf("intValue: %d\n", data.intValue);
printf("floatValue: %.2f\n", data.floatValue);
printf("stringValue: %s\n", data.stringValue);

return 0;
}

```

Difference in Structure and Union

	STRUCTURE	UNION
Keyword	The keyword struct is used to define a structure	The keyword union is used to define a union.
Size	When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is greater than or equal to the sum of sizes of its members.	when a variable is associated with a union, the compiler allocates the memory by considering the size of the largest memory. So, size of union is equal to the size of largest member.
Memory	Each member within a structure is assigned unique storage area of location.	Memory allocated is shared by individual members of union.
Value Altering	Altering the value of a member will not affect other members of the structure.	Altering the value of any of the member will alter other member values.
Accessing members	Individual member can be accessed at a time.	Only one member can be accessed at a time.
Initialization of Members	Several members of a structure can initialize at once.	Only the first member of a union can be initialized.
