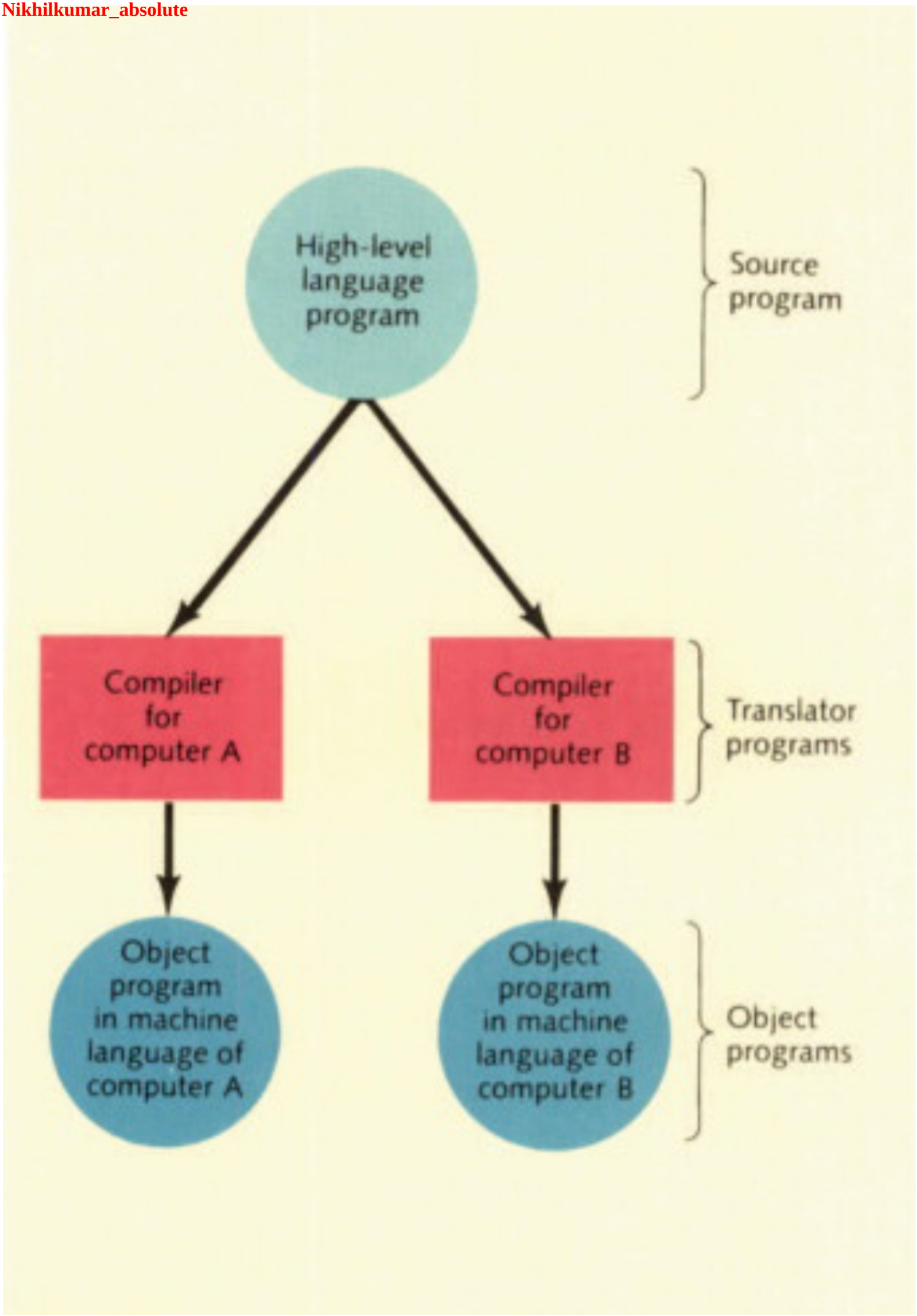
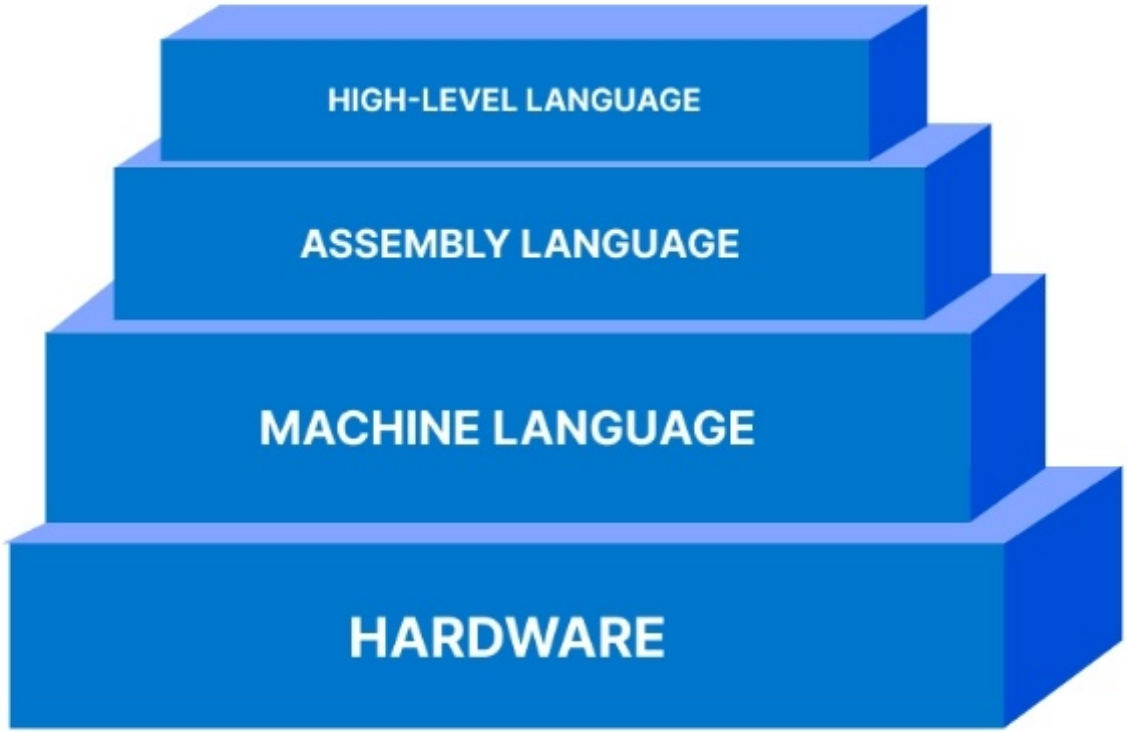
Join here



High-Level Language (HHL)



Machine Languages

- Lowest-level programming language
- Consists of binary digits (0 and 1)
- Directly understood by the computer's processor
- Difficult for humans to read and write

Assembly Languages

- Low-level programming language
- Uses mnemonic codes (e.g., ADD, SUB, MOV) instead of binary digits
- Provides a one-to-one correspondence with machine language instructions
- Easier to read and write compared to machine language
- Requires an assembler to translate into machine language

High-Level Programming Languages

- Closer to human language and easier to read and write
- Examples: Python, Java, C++, JavaScript, Ruby
- Provide abstraction from the underlying hardware
- Require a compiler or interpreter to translate into machine language

Algorithms and Flowcharts

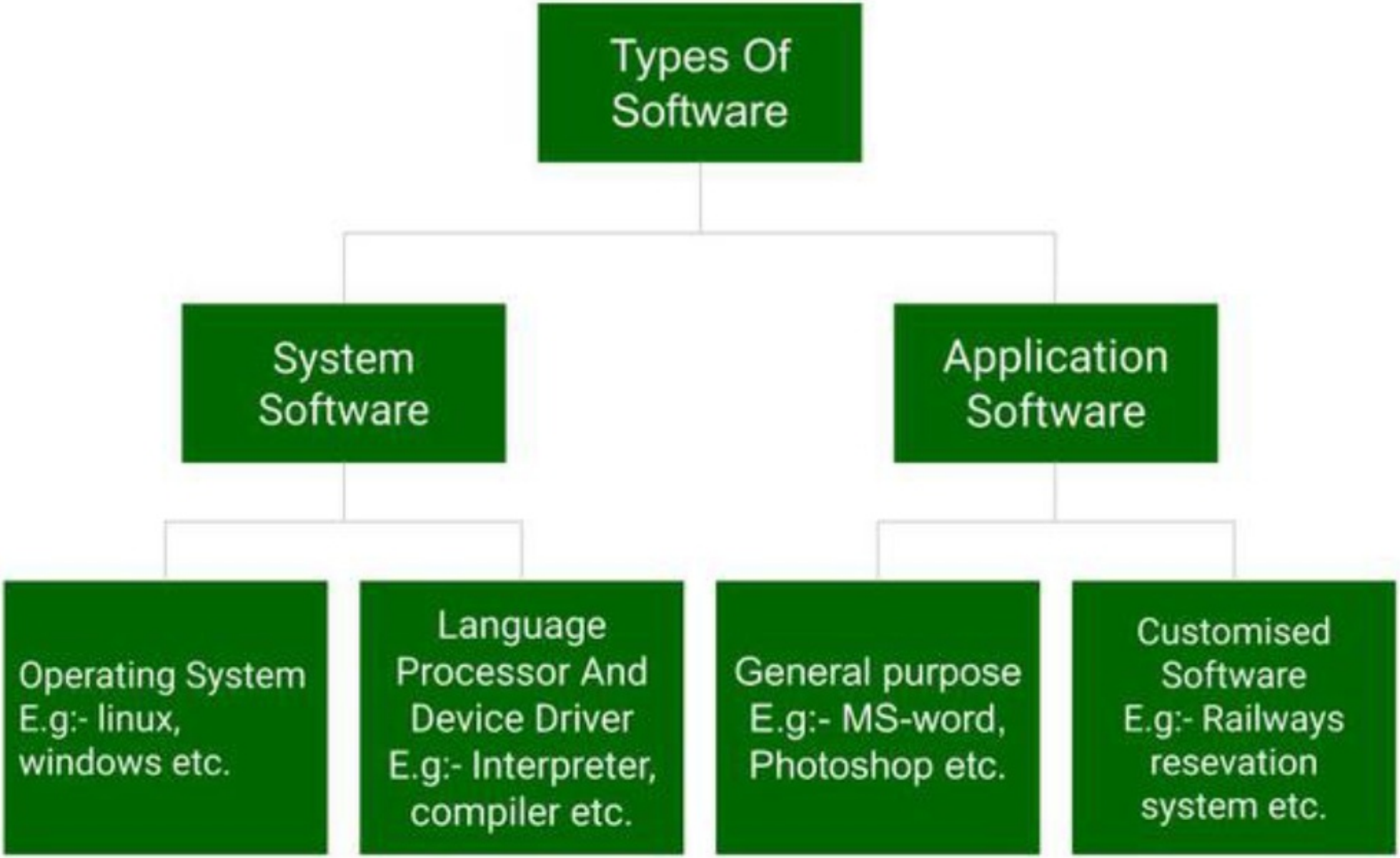
Algorithms are step-by-step procedures for solving a problem or performing a task. They can be represented using:

Flowcharts

- Graphical representations of an algorithm
- Use various symbols (e.g., rectangles, diamonds, ovals) to represent different steps
- Arrows indicate the flow of control and the order of execution
- Helps visualize the logic and structure of an algorithm

Pseudocode

- An informal description of an algorithm using a combination of natural language and programming language constructs
- Provides a high-level, abstract representation of an algorithm
- Helps express the logic and structure of an algorithm without getting into the details of a specific programming language


Types Of Software
System Software
Application Software
Operating System E.g:- linux, windows etc.
Language Processor And Device Driver E.g:- Interpreter, compiler etc.
General purpose E.g:- MS-word, Photoshop etc.
Customised Software E.g:- Railways resevation system etc.

Introduction to Computer Software

Classification of Computer Software

Computer software can be broadly classified into two main categories:

1. **System Software**
    - Provides an interface between the computer hardware and the user
    - Examples: Operating systems (e.g., Windows, macOS, Linux), device drivers, utility programs
2. **Application Software**
    - Designed to perform specific tasks for the user
    - Examples: Word processors (e.g., Microsoft Word), spreadsheets (e.g., Microsoft Excel), web browsers (e.g., Google Chrome), media players (e.g., VLC)

Programming Languages

Programming languages are used to write computer programs and software. They provide a way for humans to communicate with computers and give instructions.

Some popular programming languages include:

- **Python**: A high-level, general-purpose language known for its simplicity and readability
- **Java**: A class-based, object-oriented language widely used for developing applications and web services
- **C++**: An extension of the C programming language that adds object-oriented features
- **JavaScript**: A scripting language primarily used for web development and creating interactive web pages
- **Ruby**: A dynamic, object-oriented language known for its simplicity and readability

Programming languages can be classified based on their level of abstraction:

- **High-level languages**: Closer to human language and easier to read and write (e.g., Python, Java, C++)
- **Low-level languages**: Closer to the machine language and more difficult to read and write (e.g., assembly language)


BIHAR
DIPLOMA
UNIT-2
[All materials]
t.me/patna7diploma
@nikhilkumar_absolute

Fundamentals of C Language

Introduction

C is a general-purpose programming language that has had a significant impact on many modern programming languages. It is known for its efficiency and flexibility, making it a popular choice for system and application development.

Background

- Developed in the early 1970s by Dennis Ritchie at Bell Labs.
- Originally created for system programming and developing the UNIX operating system.
- Influenced many languages, including C++, C#, and Java.

Characteristics of C

- **Simplicity**: C has a straightforward syntax, making it easier to learn and use.
- **Efficiency**: Produces fast and efficient machine code, suitable for system-level programming.
- **Rich set of operators**: Supports a wide range of operators for arithmetic, logical, and bitwise operations.
- **Portability**: Programs written in C can be compiled and run on various hardware platforms with minimal changes.
- **Modularity**: Supports functions, allowing for code reuse and better organization.
- **Low-level access**: Provides features for direct manipulation of hardware and memory (e.g., pointers).

Uses of C

- **System Programming**: Developing operating systems, device drivers, and embedded systems.
- **Application Development**: Creating software applications, including databases and graphic applications.
- **Game Development**: Used in game engines and game programming for performance-critical applications.
- **Embedded Systems**: Widely used in programming microcontrollers and embedded systems due to its efficiency.

Structure of a C Program

A typical C program consists of the following components:

1. **Preprocessor Directives**: Instructions for the preprocessor (e.g., `#include`, `#define`).
2. **Main Function**: The entry point of the program, defined as `int main()`.
3. **Variable Declarations**: Declaration of variables used in the program.
4. **Function Definitions**: Definitions of user-defined functions.
5. **Statements and Expressions**: The actual code that performs operations (e.g., assignments, calculations).
6. **Return Statement**: Indicates the end of the `main` function and returns a value (e.g., `return 0;`).

Writing the First C Program

A simple "Hello, World!" program in C:

```c
#include <stdio.h> // Preprocessor directive

int main() { // Main function
    printf("Hello, World!\n"); // Print statement
    return 0; // Return statement
}
```

Files Used in a C Program

C programs typically involve the following types of files:

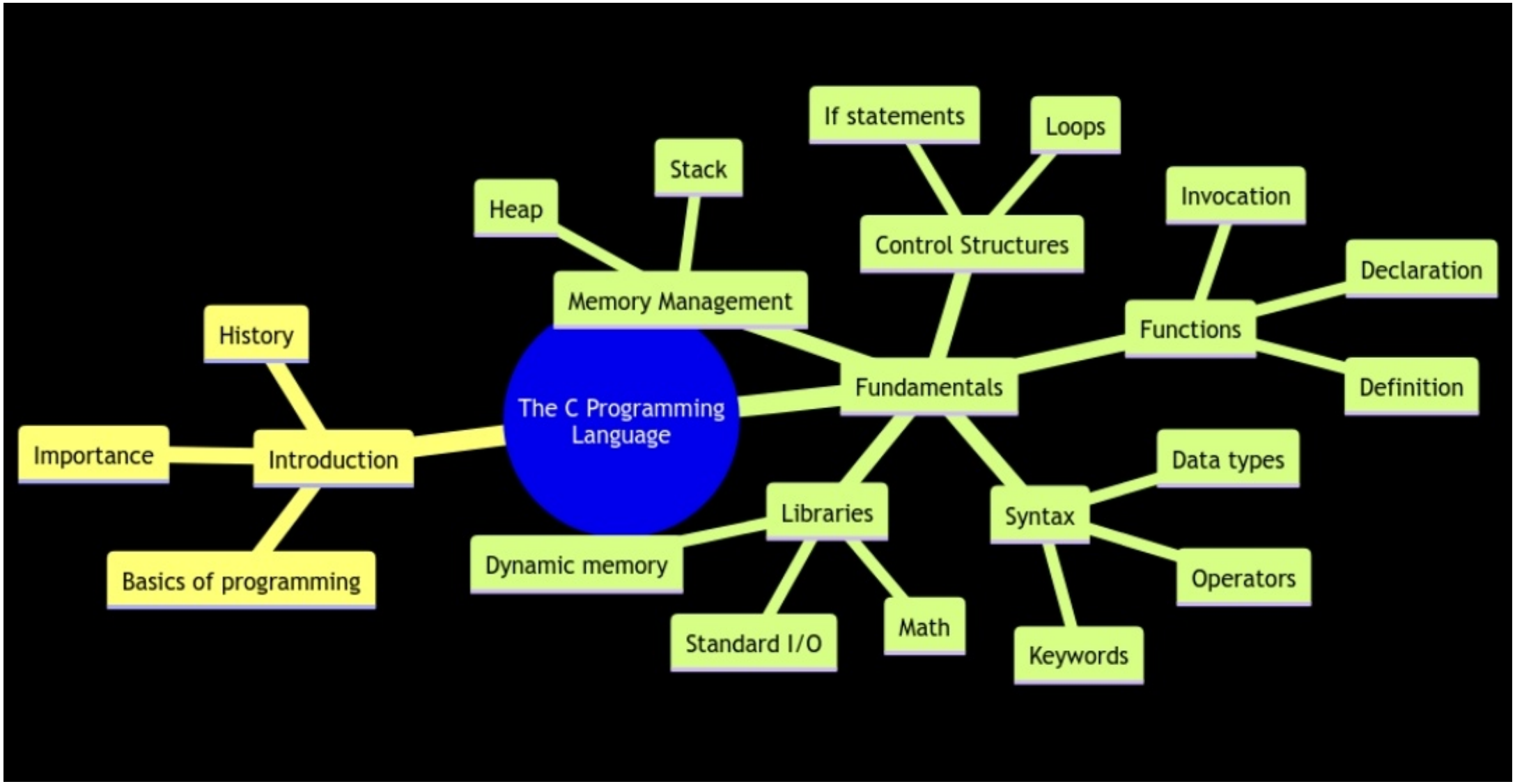
- **Source Files**: Files containing the C code, usually with a `.c` extension (e.g., `program.c`).
- **Header Files**: Files that contain function declarations and macro definitions, usually with a `.h` extension (e.g., `stdio.h`).
- **Object Files**: Compiled files generated from source files, usually with a `.o` or `.obj` extension.
- **Executable Files**: The final output of the compilation process, which can be run on the system (e.g., `program.exe` on Windows, or `./program` on Unix/Linux).

These components together form the foundation of programming in C, enabling developers to create efficient and effective software solutions.

Structure of C Program

	1	`#include <stdio.h>`	Header
	2	`int main(void)`	Main
BODY	3	`{`	
BODY	4	`printf("Hello World");`	Statement
BODY	5	`return 0;`	Return
BODY	6	`}`	

Nikhilkumar_absolute



Source Code Files

- Files containing the C code written by the programmer
- Usually have a `.c` extension (e.g., `program.c`)
- Contain function definitions, variable declarations, and statements

Header Files

- Files that contain function declarations and macro definitions
- Usually have a `.h` extension (e.g., `stdio.h`)
- Included in source code files using preprocessor directives (e.g., `#include <stdio.h>`)
- Allows for code reuse and modularization

Object Files

- Compiled files generated from source code files
- Usually have a `.o` or `.obj` extension
- Contain machine-readable code that can be linked with other object files

Binary Executable Files

- The final output of the compilation process
- Can be run on the system
- Examples: `program.exe` on Windows, `./program` on Unix/Linux

Compiling and Executing C Programs

1. **Write the source code** in a text editor and save it with a `.c` extension.
2. **Compile the source code** using a C compiler (e.g., GCC, Clang).
3. **Link the compiled object files** to create an executable.
4. **Run the executable** to execute the program.

Using Comments

- Used to add explanatory notes and descriptions in the code
- Ignored by the compiler
- Two types: single-line comments (`//`) and multi-line comments (`/* */`)

Characters Used in C

- Uppercase and lowercase letters (A-Z, a-z)
- Digits (0-9)
- Special characters (e.g., `+`, `-`, `*`, `/`, `=`, `(`, `)`, `{`, `}`, `;`, `,`, `.`)

Identifier

- Names given to variables, functions, and other entities in C
- Must start with a letter or underscore (`_`)
- Can contain letters, digits, and underscores
- Case-sensitive (e.g., `myVariable` is different from `myvariable`)

Keywords or Reserved Words

- Words that have predefined meanings in C
- Cannot be used as identifiers (e.g., `int`, `float`, `if`, `else`, `while`, `for`)

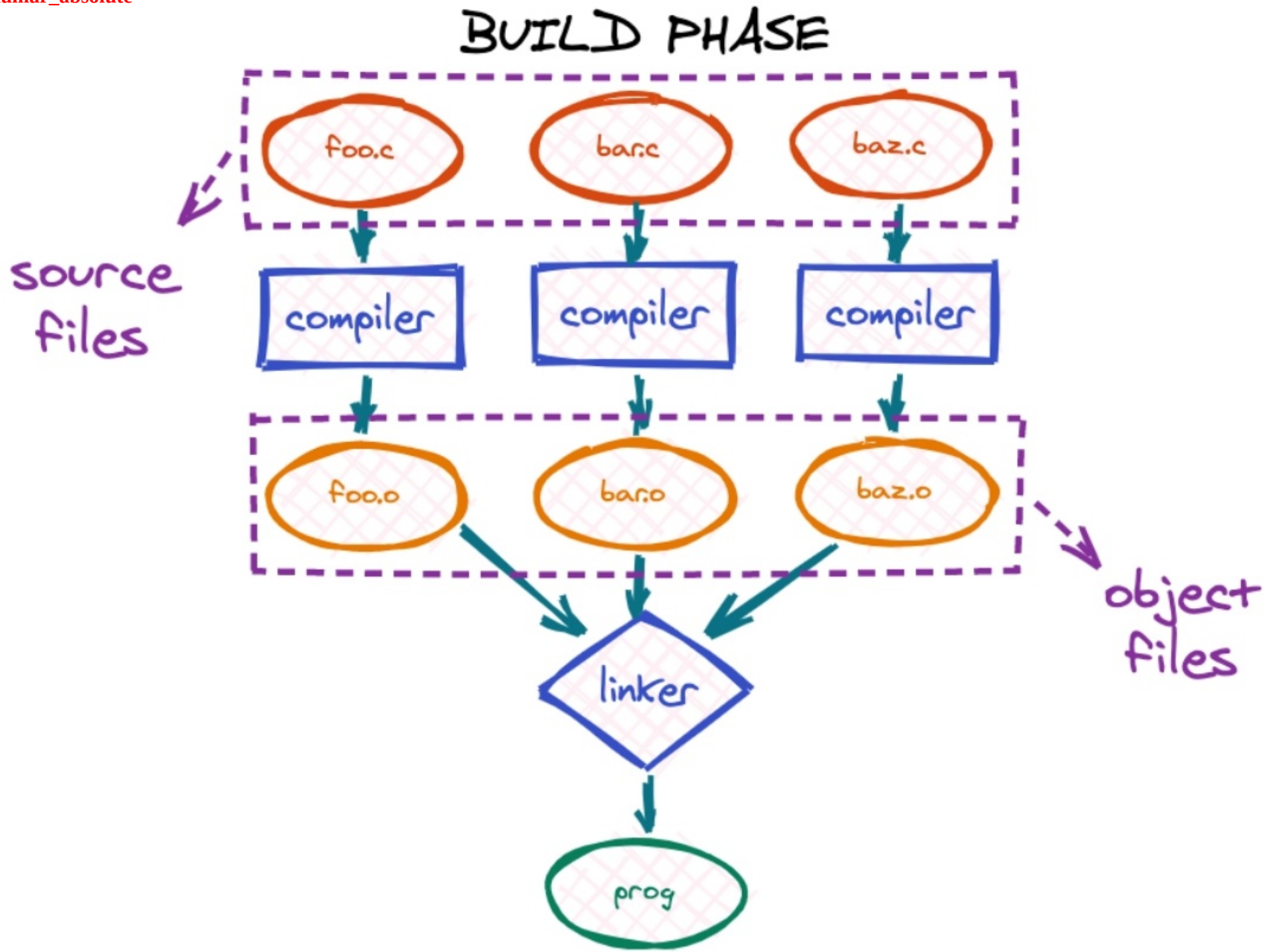
Tokens

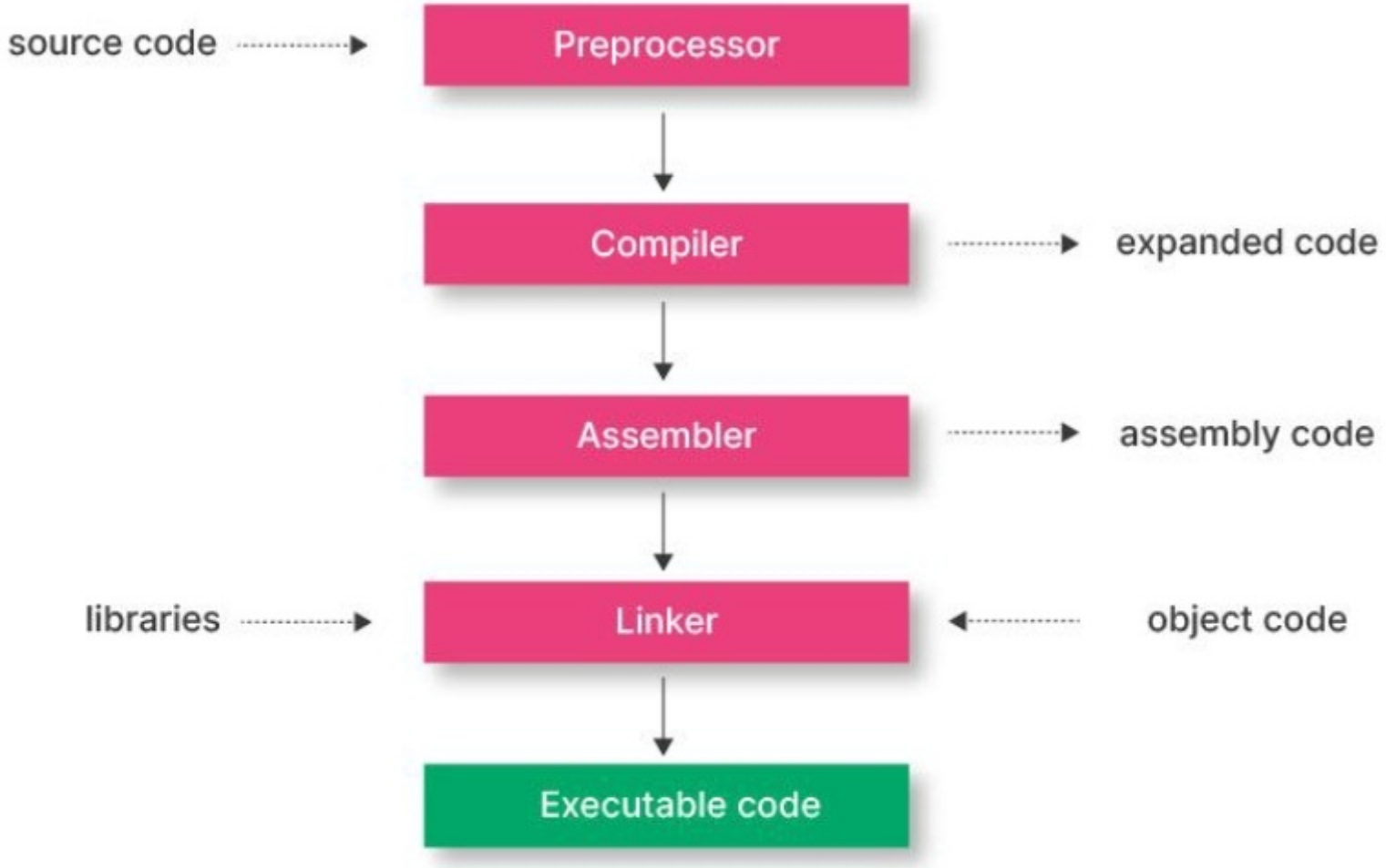
- The smallest individual units in a C program
- Examples: keywords, identifiers, constants, operators, punctuators (e.g., `{`, `}`, `;`)

Constants

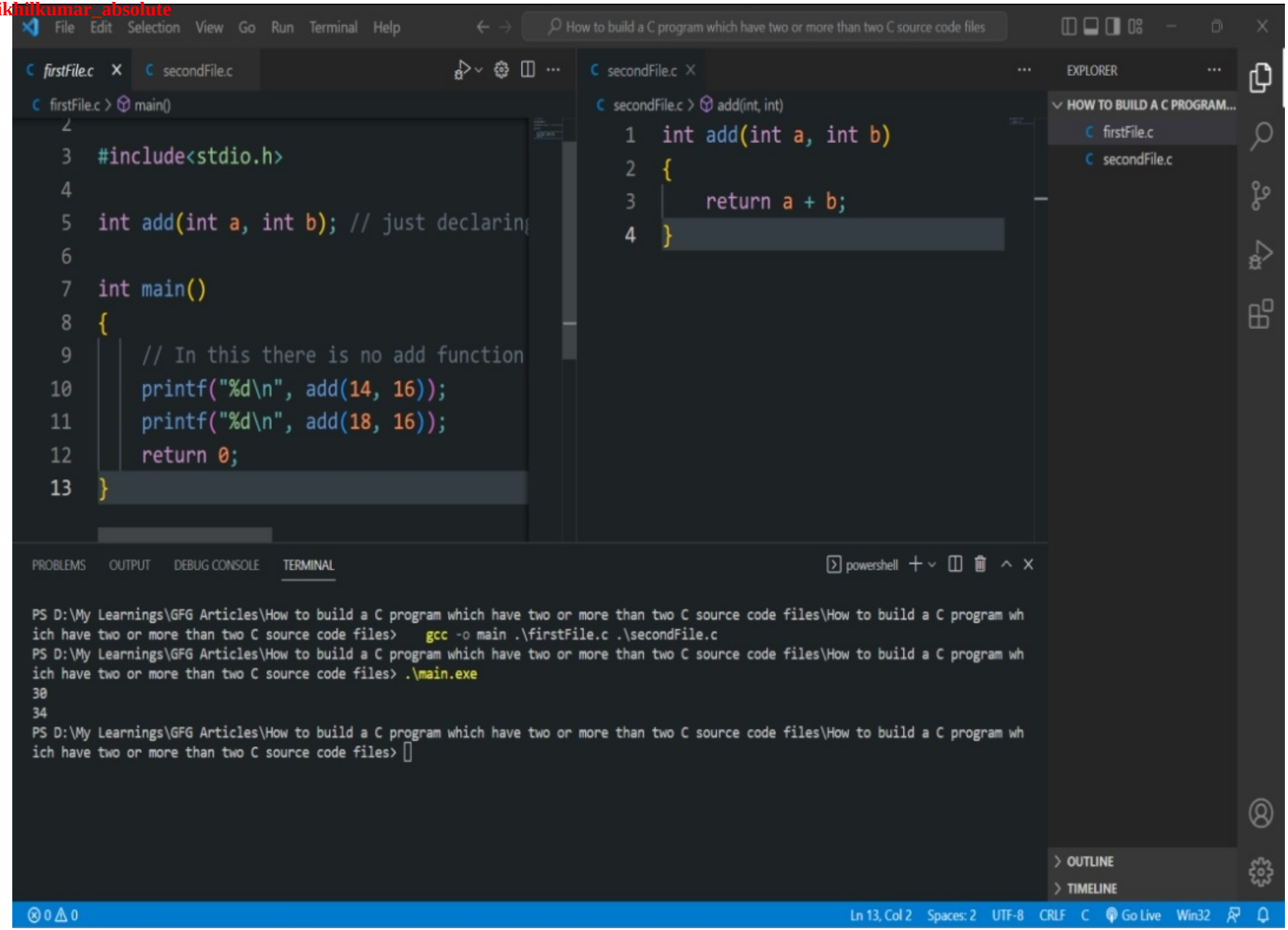
- Values that cannot be modified during program execution
- Examples: integer constants (e.g., `42`), floating-point constants (e.g., `3.14`), character constants (e.g., `'A'`), string constants (e.g., `"Hello, World!"`)

Nikhilkumar_absolute




source code
Preprocessor
Compiler
expanded code
Assembler
assembly code
libraries
Linker
object code
Executable code

Nikhilkumar_absolute



Numeric Constant

- A numeric constant is a fixed value that represents a number.
- Types of numeric constants:
  - **Integer Constants**: Whole numbers (e.g., `42`, `-7`, `0`)
  - **Floating-Point Constants**: Numbers with decimal points (e.g., `3.14`, `-0.001`, `2.0e3`)

String Character Constant

- A string character constant is a sequence of characters enclosed in double quotes.
- Represents text data (e.g., `"Hello, World!"`)
- Each character in the string is stored as an individual character constant.

Variables

- Variables are named storage locations in memory that hold data.
- Their values can change during program execution.
- Must be declared before use.

Variable Declaration

- A statement that specifies the variable's name and data type.
- Syntax: `data_type variable_name;`
- Example: `int age;` declares a variable named `age` of type `int`.

Basic Data Types

- C supports several basic data types:
  - **int**: Integer type (e.g., `-1`, `0`, `42`)
  - **float**: Single-precision floating-point type (e.g., `3.14f`)
  - **double**: Double-precision floating-point type (e.g., `2.71828`)
  - **char**: Character type (e.g., `'A'`, `'z'`)

Additional Data Types

- **void**: Represents the absence of value or type.
- **short**: Short integer type, typically 16 bits.
- **long**: Long integer type, typically 32 or 64 bits, depending on the system.
- **unsigned**: Modifier that can be applied to integer types to represent only non-negative values.

Operators and Expressions

- **Operators**: Symbols that perform operations on variables and values.
  - **Arithmetic Operators**: `+`, `-`, `*`, `/`, `%`
  - **Relational Operators**: `==`, `≠`, `<`, `>`, `≤`, `≥`
  - **Logical Operators**: `&&`, `||`, `!`
  - **Bitwise Operators**: `&`, `|`, `^`, `~`, `<<`, `>>`
  - **Assignment Operators**: `=`, `+=`, `-=`, `*=`, `/=`, `%=`
- **Expressions**: Combinations of variables, constants, and operators that produce a value.

Operator Precedence and Associativity

- **Operator Precedence**: Determines the order in which operators are evaluated in an expression.
  - Higher precedence operators are evaluated before lower precedence ones.
  - Example: In the expression `3 + 4 * 5`, multiplication is performed before addition.
- **Associativity**: Determines the order of evaluation for operators of the same precedence.
  - Left-to-right (e.g., `-` and `+`): `a - b + c`
  - Right-to-left (e.g., `=`): `a = b = c`

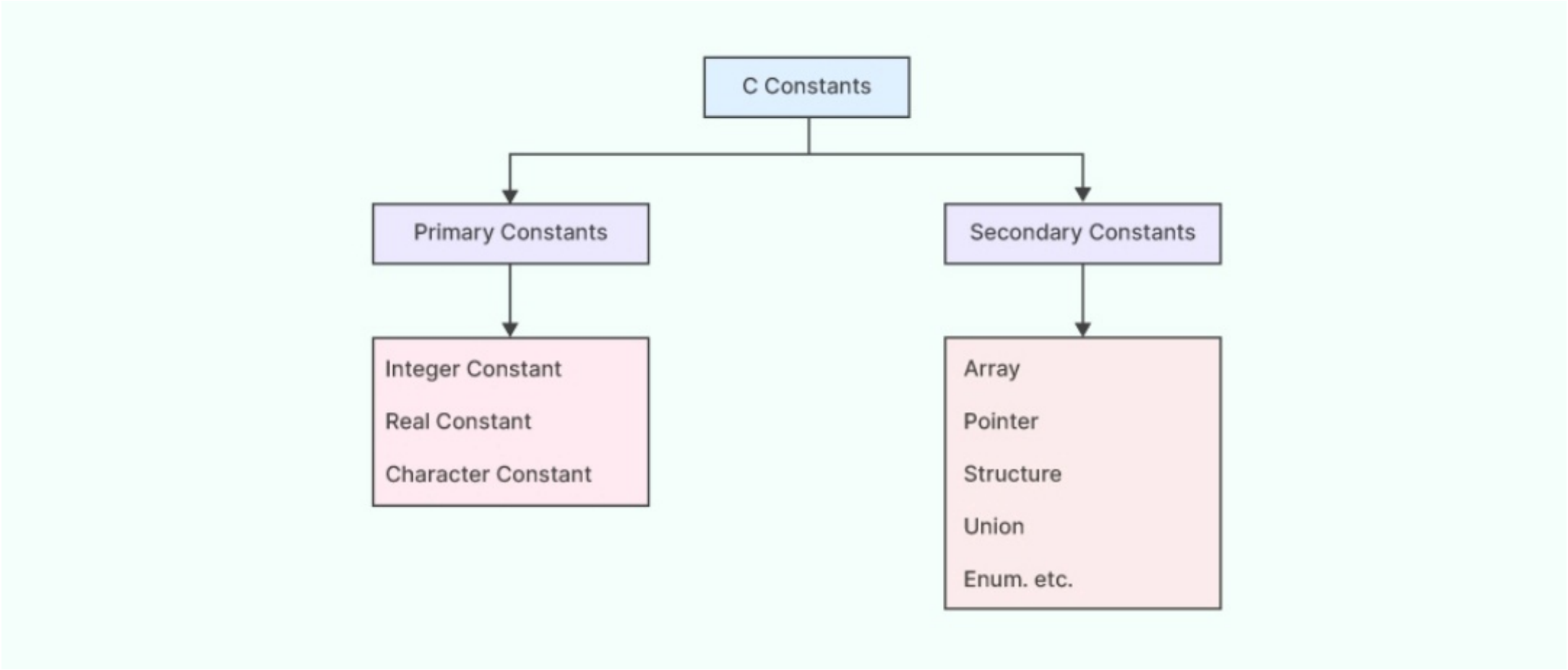
Type Conversion and Type Casting

- **Type Conversion**: Automatic conversion of one data type to another by the compiler.
  - Example: Assigning an `int` to a `float` variable.
- **Type Casting**: Explicit conversion of one data type to another using casting operators.
  - Syntax: `(data_type) expression`
  - Example: `float f = (float)intVar;`

Input/Output Statements in C

- **Input**: Reading data from the user using the `scanf` function.
  - Syntax: `scanf("format_specifier", &variable);`
  - Example: `scanf("%d", &age);` reads an integer input into the variable `age`.
- **Output**: Displaying data to the user using the `printf` function.
  - Syntax: `printf("format_string", variables);`
  - Example: `printf("Age: %d\n", age);` prints the value of `age`.

Nikhilkumar_absolute




BIHAR
DIPLOMA
UNIT-3
[All materials]
t.me/patna7diploma
@nikhilkumar_absolute

Introduction to Decision Control Statements

Decision control statements allow you to execute different blocks of code based on certain conditions. They enable you to make choices and control the flow of execution in your program.

Conditional Branching Statements

If Statement

- The simplest decision control statement in C
- Executes a block of code if a given condition is true
- Syntax:

```c
if (condition) {
    // Code block to be executed if condition is true
}
```

If-Else Statement

- Executes one block of code if the condition is true and another block if the condition is false
- Syntax:

```c
if (condition) {
    // Code block to be executed if condition is true
} else {
    // Code block to be executed if condition is false
}
```

If-Else-If Statement

- Allows you to check multiple conditions
- Executes the first block of code with a true condition
- If none of the conditions are true, it executes the optional `else` block
- Syntax:

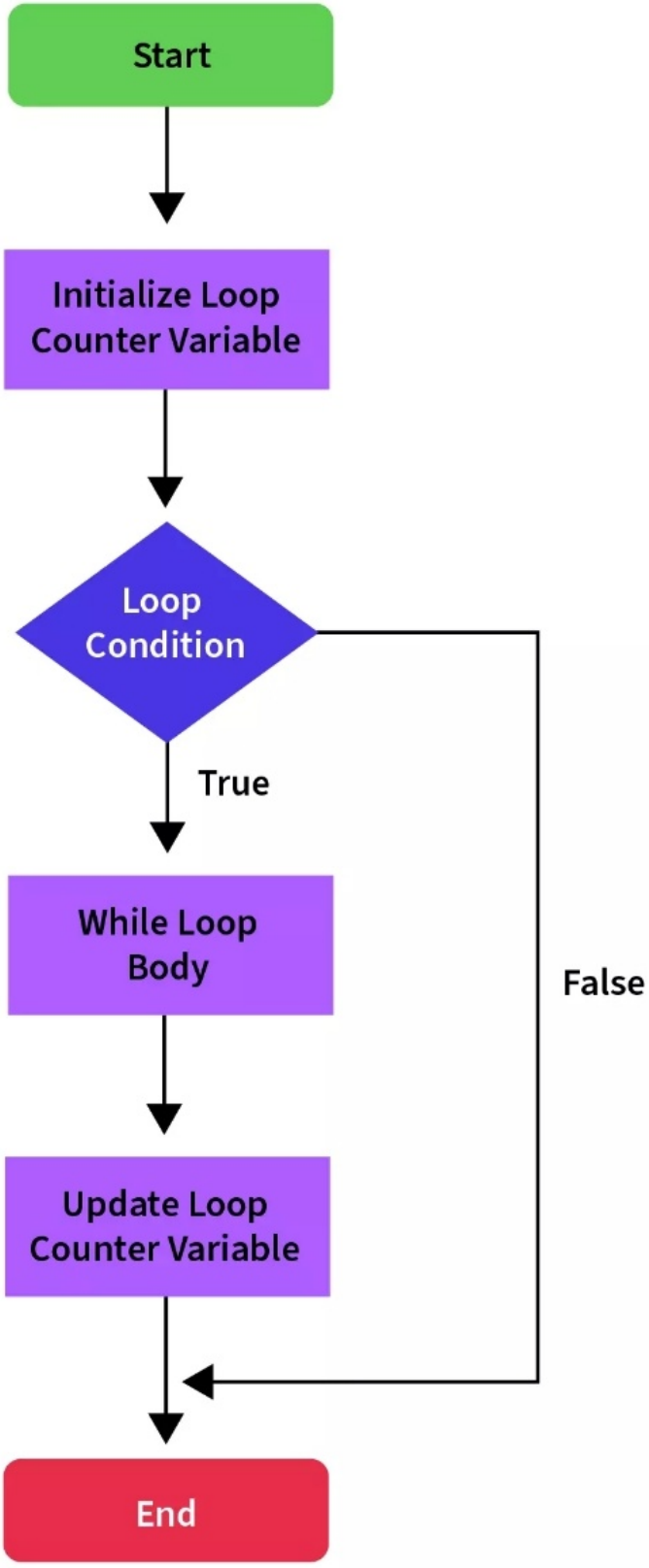
```c
if (condition1) {
    // Code block to be executed if condition1 is true
} else if (condition2) {
    // Code block to be executed if condition2 is true
} else if (condition3) {
    // Code block to be executed if condition3 is true
} else {
    // Code block to be executed if all conditions are false
}
```

Switch Case

- Provides an alternative to multiple `if-else` statements
- Executes a block of code based on the value of a single expression
- Syntax:

```c
switch (expression) {
    case value1:
        // Code block to be executed if expression matches value1
        break;
    case value2:
        // Code block to be executed if expression matches value2
        break;
    ...
    default:
        // Code block to be executed if expression doesn't match any case
}
```

These decision control statements allow you to write more complex and flexible programs by making choices based on specific conditions. They are essential for creating programs that can adapt to different scenarios and user inputs.


Start
Initialize Loop Counter Variable
Loop Condition
True
False
While Loop Body
Update Loop Counter Variable
End

Iterative Statements

Iterative statements allow you to execute a block of code repeatedly based on a specified condition. They are essential for performing repetitive tasks efficiently.

While Loop

- Executes a block of code as long as a specified condition is true.
- The condition is checked before each iteration.
- Syntax:

```c
while (condition) {
    // Code block to be executed
}
```

Do-While Loop

- Similar to the `while` loop, but the condition is checked after the code block is executed.
- Guarantees that the code block executes at least once.
- Syntax:

```c
do {
    // Code block to be executed
} while (condition);
```

For Loop

- A more compact way to write loops, especially when the number of iterations is known.
- Combines initialization, condition checking, and increment/decrement in one line.
- Syntax:

```c
for (initialization; condition; increment/decrement) {
    // Code block to be executed
}
```

- Example:

```c
for (int i = 0; i < 10; i++) {
    // Code block to be executed
}
```

Nested Loops

- A loop inside another loop.
- The inner loop executes completely for each iteration of the outer loop.
- Useful for multi-dimensional data structures (e.g., matrices).
- Example:

```c
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 2; j++) {
        // Code block to be executed
    }
}
```

Break and Continue Statements

Break Statement

- Used to exit a loop or switch statement prematurely.
- When encountered, it immediately terminates the loop or switch and transfers control to the next statement following the loop or switch.
- Example:

```c
while (condition) {
    if (some_condition) {
        break; // Exit the loop
    }
}
```

Continue Statement

- Skips the current iteration of a loop and proceeds to the next iteration.
- When encountered, it immediately jumps to the next iteration of the loop, skipping any remaining code in the current iteration.
- Example:

```c
for (int i = 0; i < 10; i++) {
    if (i % 2 == 0) {
        continue; // Skip even numbers
    }
    // Code block for odd numbers
}
```

Goto Statement

- A control statement that allows for an unconditional jump to another part of the program.
- Generally discouraged in structured programming due to potential for creating "spaghetti code," making programs harder to read and maintain.
- Syntax:

```c
goto label;

label:
    // Code block to be executed
```

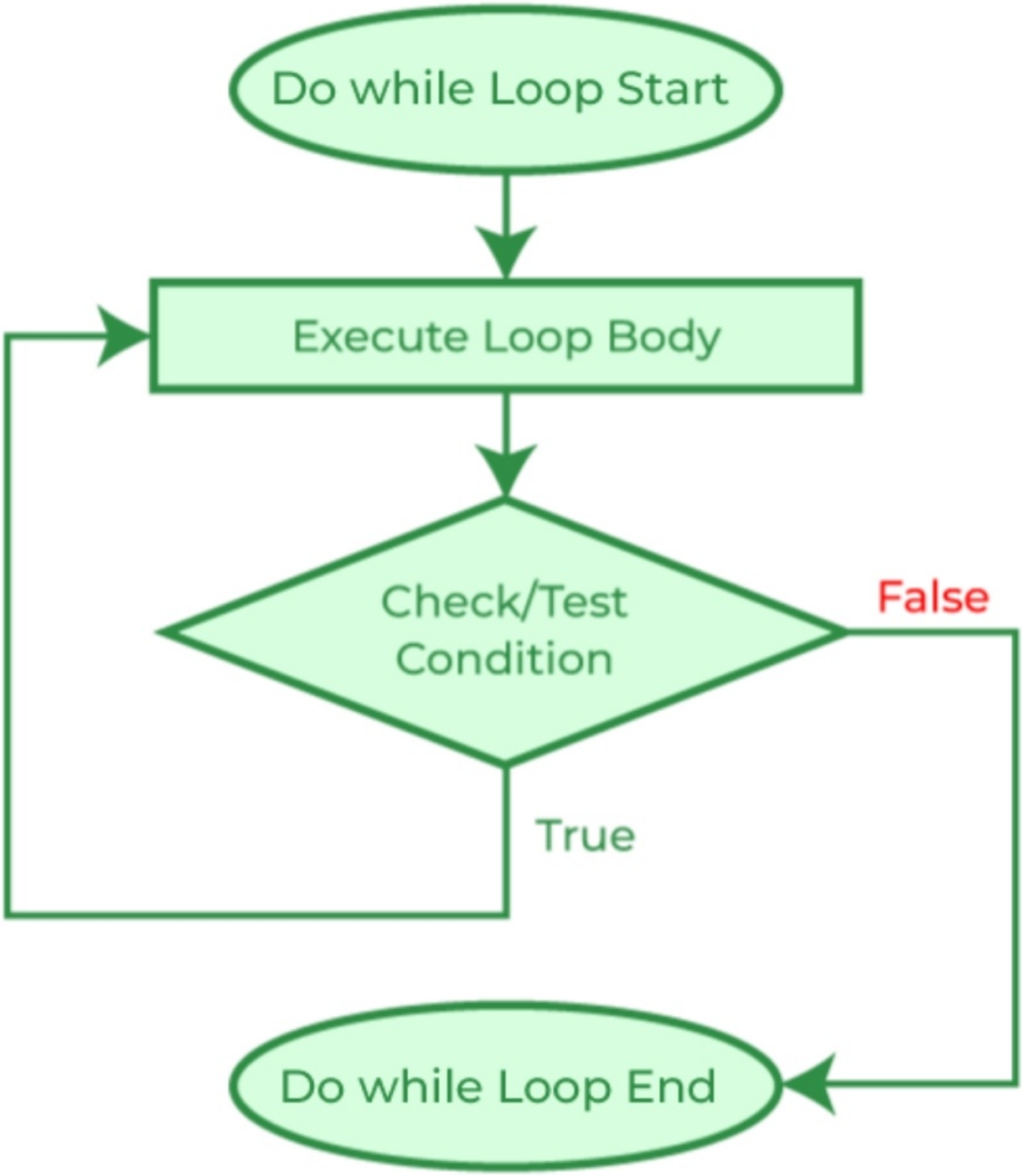
- Example:

```c
if (condition) {
    goto end; // Jump to the end label
}

// Other code

end:
    // Code block to be executed after the jump
```

These iterative statements and control flow mechanisms are fundamental in C programming, allowing for efficient execution of repetitive tasks and enabling complex program logic.


Do while Loop Start
Execute Loop Body
Check/Test Condition
False
True
Do while Loop End


BIHAR
DIPLOMA
UNIT-4
[All materials]
t.me/patna7diploma
@nikhilkumar_absolute

Functions in C

Functions are self-contained blocks of code that perform a specific task. They help in organizing code, promoting reuse, and improving readability.

Uses of Functions

- **Modularity**: Breaks down complex problems into smaller, manageable pieces.
- **Code Reusability**: Allows the same function to be used multiple times without rewriting code.
- **Ease of Maintenance**: Changes can be made in one place without affecting the entire program.
- **Improved Readability**: Functions can make code easier to understand by encapsulating specific tasks.

User Defined Functions

- Functions defined by the programmer to perform specific tasks.
- Can take parameters and return values.
- Syntax:

```c
return_type function_name(parameter_list) {
    // Code block
    return value; // Optional
}
```

- Example:

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

Function Declaration

- A statement that specifies the function's name, return type, and parameters without defining the function body.
- Also known as a function prototype.
- Syntax:

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

- Example:

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

Calling a Function

- To execute a function, you call it by its name and provide the required arguments.
- Syntax:

```c
function_name(arguments);
```

- Example:

```c
int result = add(5, 10); // Calls the add function
```

Actual and Formal Arguments

- **Actual Arguments**: The values or variables passed to a function when it is called.
  - Example: In `add(5, 10)`, `5` and `10` are actual arguments.
- **Formal Arguments**: The parameters defined in the function declaration or definition.
  - Example: In `int add(int a, int b)`, `a` and `b` are formal arguments.

Rules to Call a Function

1. The function must be declared before it is called.
2. The number and types of actual arguments must match the formal parameters in the function declaration.
3. The return type of the function should be compatible with the variable receiving the return value.

Function Prototype

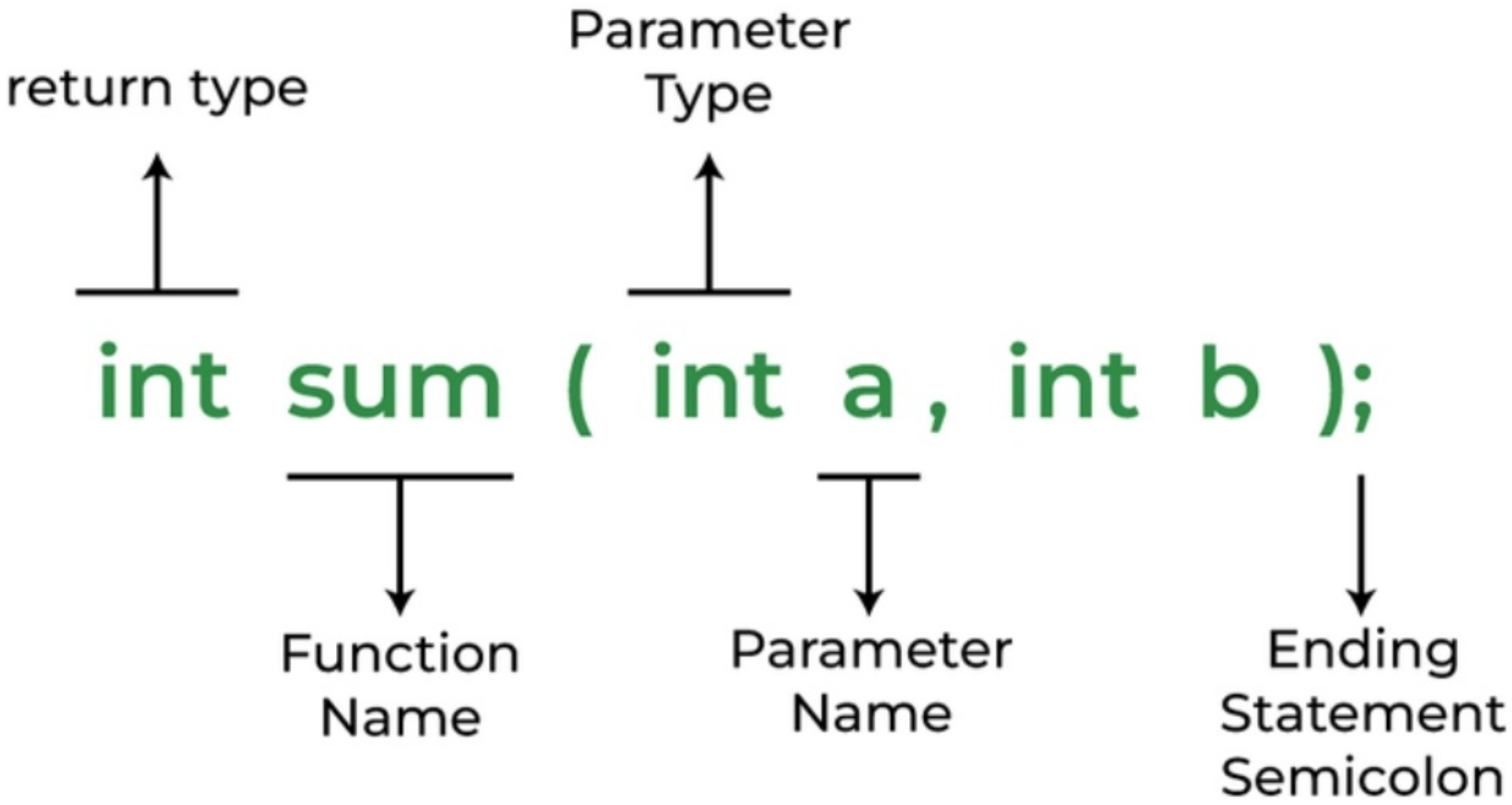
- A declaration of a function that specifies its name, return type, and parameters.
- Helps the compiler to understand the function's interface before its actual definition.
- Example:

```c
float calculateArea(float radius);
```

Recursion

- A process in which a function calls itself directly or indirectly.
- Used for solving problems that can be broken down into smaller, similar subproblems.
- Must have a **base case** to terminate the recursion and prevent infinite loops.
- Example:

```c
int factorial(int n) {
    if (n == 0) // Base case
        return 1;
    else
        return n * factorial(n - 1); // Recursive call
}
```


return type
Parameter Type
int sum (int a , int b);
Function Name
Parameter Name
Ending Statement Semicolon

Use of Recursive Functions

- Recursion is a programming technique where a function calls itself to solve a problem.
- It is useful for solving problems that can be broken down into smaller, similar subproblems.
- Recursive functions must have a base case to terminate the recursion and prevent infinite loops.
- Example: Calculating the factorial of a number using recursion:

```c
int factorial(int n) {
    if (n == 0) // Base case
        return 1;
    else
        return n * factorial(n - 1); // Recursive call
}
```

Local or Internal Variables

- Variables declared within a function or block.
- Have local scope and are only accessible within the block where they are defined.
- Exist only during the execution of the block and are destroyed when the block ends.

Global or External Variables

- Variables declared outside any function or block.
- Have global scope and can be accessed from any part of the program.
- Exist throughout the entire program's execution.
- Initialized to zero by default.

Void Function

- A function that does not return any value.
- Declared with the `void` keyword as the return type.
- Can still take parameters and perform operations.
- Example:

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

Storage Classes in C

Storage classes in C determine the scope, lifetime, and initial value of variables and functions.

Auto or Automatic Storage Class

- The default storage class for variables declared inside a function or block.
- Variables have local scope and are destroyed when the block ends.
- Initialized with garbage values.

Static Storage Class

- Variables retain their values between function calls.
- Variables have local scope within the file or block where they are defined.
- Initialized to zero by default.
- Example:

```c
static int count = 0;
```

Extern Storage Class

- Used to access variables or functions declared in other files.
- Allows sharing variables or functions between multiple source files.
- Declared with the `extern` keyword.
- Example:

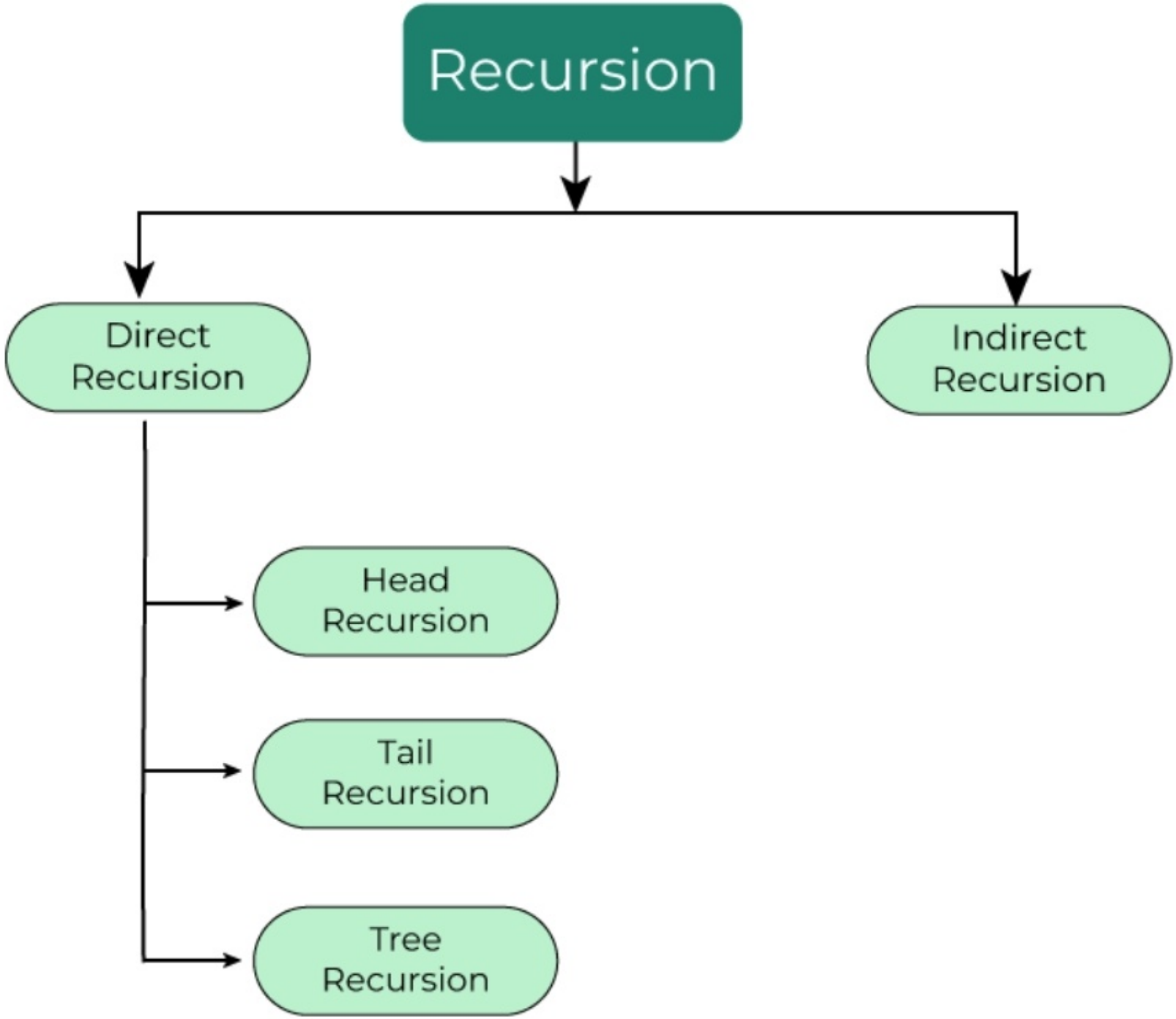
```c
extern int globalVar;
```

Register Storage Class

- Suggests to the compiler that the variable should be stored in a CPU register for faster access.
- The compiler may or may not honor the request based on availability and other factors.
- Variables have local scope and are destroyed when the block ends.
- Example:

```c
register int count;
```

Nikhilkumar_absolute




BIHAR
DIPLOMA
UNIT-5
[All materials]
t.me/patna7diploma
@nikhilkumar_absolute

Arrays

Introduction

- An array is a collection of elements of the same data type stored in contiguous memory locations.
- Arrays allow for efficient data management and manipulation, enabling the storage and processing of multiple values using a single variable name.
- Commonly used for storing lists of items, such as numbers or characters.

Declaration of Arrays

- An array must be declared before it can be used. The declaration specifies the array's name, type, and size.
- Syntax:

```c
data_type array_name[array_size];
```

- Example:

```c
int numbers[5]; // Declares an array of 5 integers
```

Accessing the Elements of an Array

- Array elements are accessed using their index, which starts at 0.
- Syntax for accessing an element:

```c
array_name[index]
```

- Example:

```c
int firstNumber = numbers[0]; // Accesses the first element of the array
```

Calculating the Address of Array Elements

- The address of an array element can be calculated using the formula:

```text
Address of array[i] = base_address + (i * size_of_element)
```

- `base_address` is the address of the first element, `i` is the index, and `size_of_element` is the size of the data type in bytes.

Calculating the Length of an Array

- The length of a statically declared array can be calculated using the formula:

```text
length = sizeof(array) / sizeof(array[0])
```

- This gives the total number of elements in the array.

Storing Values in Arrays

Initializing Arrays During Declaration

- Arrays can be initialized at the time of declaration using curly braces.
- Syntax:

```c
data_type array_name[array_size] = {value1, value2, ..., valueN};
```

- Example:

```c
int numbers[5] = {1, 2, 3, 4, 5}; // Initializes an array with 5 integers
```

Inputting Values from the Keyboard

- Values can be inputted into an array using a loop and the `scanf` function.
- Example:

```c
for (int i = 0; i < 5; i++) {
    printf("Enter number %d: ", i + 1);
    scanf("%d", &numbers[i]); // Inputs values into the array
 }
```

Assigning Values to Individual Elements

- Individual elements of an array can be assigned values after declaration.
- Syntax:

```c
array_name[index] = value;
```

- Example:

```c
numbers[0] = 10; // Assigns the value 10 to the first element of the array
```

These concepts provide a comprehensive understanding of arrays in C programming, enabling effective data storage and manipulation techniques.

Initializing an Array in C

- We can initialize an array in C, by using an initializer list. This sets the respective elements to a corresponding element of the list.
- We can also use a simple for loop to manually set the elements of the array.
- We can also use designated initializer lists, to selectively initialize a range of elements, if we are using the gcc compiler.

```
int arr[4] = {10, 10, 10, 10};
```

Nikhilkumar_absolute

Operations on Arrays

Traversing an Array

- Traversing refers to accessing each element of the array sequentially.
- Typically done using a loop (for, while).
- Example:

```c
for (int i = 0; i < size; i++) {
    printf("%d ", array[i]); // Prints each element of the array
}
```

Inserting an Element in an Array

- Inserting involves adding a new element at a specified position.
- Steps:
  1. Shift elements to the right to create space.
  2. Insert the new element at the desired index.
- Example:

```c
void insert(int array[], int *size, int element, int position) {
    for (int i = *size; i > position; i--) {
        array[i] = array[i - 1]; // Shift elements
    }
    array[position] = element; // Insert the new element
    (*size)++; // Increment the size
}
```

Deleting an Element from an Array

- Deleting involves removing an element from a specified position.
- Steps:
  1. Shift elements to the left to fill the gap.
  2. Decrease the size of the array.
- Example:

```c
void delete(int array[], int *size, int position) {
    for (int i = position; i < *size - 1; i++) {
        array[i] = array[i + 1]; // Shift elements
    }
    (*size)--; // Decrement the size
}
```

Merging Two Arrays

- Merging combines two arrays into a single array.
- The resulting array contains elements from both arrays.
- Example:

```c
void merge(int array1[], int size1, int array2[], int size2, int merged[]) {
    for (int i = 0; i < size1; i++) {
        merged[i] = array1[i]; // Copy first array
    }
    for (int j = 0; j < size2; j++) {
        merged[size1 + j] = array2[j]; // Copy second array
    }
}
```

Searching for a Value in an Array

- Searching involves finding a specific value within an array.
- Common methods include:
  - **Linear Search**: Checks each element sequentially.

```c
int linearSearch(int array[], int size, int value) {
    for (int i = 0; i < size; i++) {
        if (array[i] == value) {
            return i; // Return index if found
        }
    }
    return -1; // Return -1 if not found
}
```

  - **Binary Search**: Requires a sorted array and divides the search interval in half.

Passing Arrays to Functions

- Arrays can be passed to functions by passing the array name (which acts as a pointer to the first element).
- Example:

```c
void printArray(int array[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", array[i]);
    }
}
```

Two-Dimensional Arrays

Declaring Two-Dimensional Arrays

- A two-dimensional array is essentially an array of arrays.
- Syntax:

```c
data_type array_name[row_size][column_size];
```

- Example:

```c
int matrix[3][4]; // Declares a 3x4 two-dimensional array
```

Initializing Two-Dimensional Arrays

- Can be initialized at the time of declaration using nested braces.
- Example:

```c
int matrix[3][4] = {
    {1, 2, 3, 4},
    {5, 6, 7, 8},
    {9, 10, 11, 12}
};
```

Accessing the Elements of Two-Dimensional Arrays

- Access elements using two indices: one for the row and one for the column.
- Syntax:

```c
array_name[row_index][column_index]
```

- Example:

```c
int value = matrix[1][2]; // Accesses the element in the second row, third column
```

Operations on Two-Dimensional Arrays

- **Traversing**: Loop through each row and column to access elements.
- **Inserting**: Can be complicated; typically involves shifting rows or columns.
- **Deleting**: Similar to inserting; may require shifting elements.
- **Matrix Addition**: Adding corresponding elements of two matrices.
- **Matrix Multiplication**: More complex; involves the dot product of rows and columns.


BIHAR
DIPLOMA
UNIT-6
[All materials]
t.me/patna7diploma
@nikhilkumar_absolute

Pointers

Understanding the Computer's Memory

- Memory in a computer is organized into bytes, each with a unique address.
- Each variable in a program is stored at a specific memory address.
- Understanding memory layout is crucial for effective use of pointers.

Introduction to Pointers

- A pointer is a variable that stores the memory address of another variable.
- Pointers provide a way to directly access and manipulate memory locations.
- They are powerful tools for dynamic memory allocation, arrays, and function arguments.

Declaring Pointer Variables

- Pointers are declared using the asterisk (`*`) symbol.
- Syntax:

```c
data_type *pointer_name;
```

- Example:

```c
int *ptr; // Declares a pointer to an integer
```

Pointer Expressions and Pointer Arithmetic

- Pointers can be manipulated using arithmetic operations.
- **Pointer Arithmetic**:
  - Incrementing a pointer (`ptr++`) moves it to the next memory location based on the data type size.
  - Example: If `ptr` points to an integer and is incremented, it will point to the next integer (4 bytes ahead on most systems).
- **Dereferencing**: Accessing the value at the address stored in a pointer using the asterisk (`*`).
  - Example:

```c
int value = *ptr; // Gets the value at the address stored in ptr
```

Null Pointers

- A null pointer is a pointer that does not point to any valid memory location.
- It is typically initialized to `NULL` to indicate that it is not currently assigned.
- Example:

```c
int *ptr = NULL; // Declares a null pointer
```

- Using null pointers can help prevent undefined behavior and segmentation faults.

Passing Arguments to Functions Using Pointers

- Pointers can be used to pass arguments to functions by reference, allowing the function to modify the original variable.
- This is useful for modifying large data structures without copying them.
- Example:

```c
void increment(int *num) {
    (*num)++; // Dereferences the pointer to increment the value
}

int main() {
    int value = 5;
    increment(&value); // Passes the address of value
    printf("%d", value); // Outputs 6
}
```

Pointer

A pointer is a variable that stores the address of another variable. In simple words, a pointer is a variable that contains the memory address of another variable.

Pointers are used in many different programming languages, including C, C++, and Java. They are powerful tools that can be used to access and manipulate data in memory.

Here are some notes about pointers:

- ➢ Pointers are variables that store memory addresses.
- ➢ Pointers can be used to access the data that is stored at the memory address that they point to.
- ➢ Pointers can be used to manipulate the data that is stored at the memory address that they point to.
- ➢ Pointers can be used to dynamically allocate memory.
- ➢ Pointers can be used to create data structures such as linked lists and trees.

Here are some examples of how pointers can be used:

- ➢ To access the data that is stored at a memory address:

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

// The value of x is now 10.
```

- ➢ To manipulate the data that is stored at a memory address:

```
int x = 10;
```

Pointers and Arrays

- Arrays and pointers are closely related in C.
- The name of an array is a constant pointer to the first element of the array.
- Array elements can be accessed using both array indexing and pointer arithmetic.
- Example:

```c
int arr[] = {1, 2, 3, 4};
printf("%d %d", arr[2], *(arr + 2)); // Both print 3
```

Passing an Array to a Function

- When an array is passed to a function, only the pointer to the first element is passed.
- The function can access and modify the array elements using the pointer.
- Example:

```c
void printArray(int *arr, int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
}
```

Dynamic Memory Allocation

Dynamic memory allocation allows programs to allocate memory at runtime, providing flexibility in memory management.

Malloc() Function

- Allocates a block of memory of the specified size and returns a pointer to the allocated memory.
- Syntax:

```c
ptr = (cast-type*) malloc(byte-size)
```

- Example:

```c
int *ptr = (int*) malloc(sizeof(int) * 5);
```

Calloc() Function

- Allocates a block of memory for an array of elements of the specified size and initializes them to zero.
- Syntax:

```c
ptr = (cast-type*) calloc(element-count, element-size)
```

- Example:

```c
int *ptr = (int*) calloc(5, sizeof(int));
```

Realloc() Function

- Changes the size of the memory block pointed to by the pointer.
- Syntax:

```c
ptr = realloc(ptr, new-size)
```

- Example:

```c
ptr = realloc(ptr, sizeof(int) * 10);
```

Free() Function

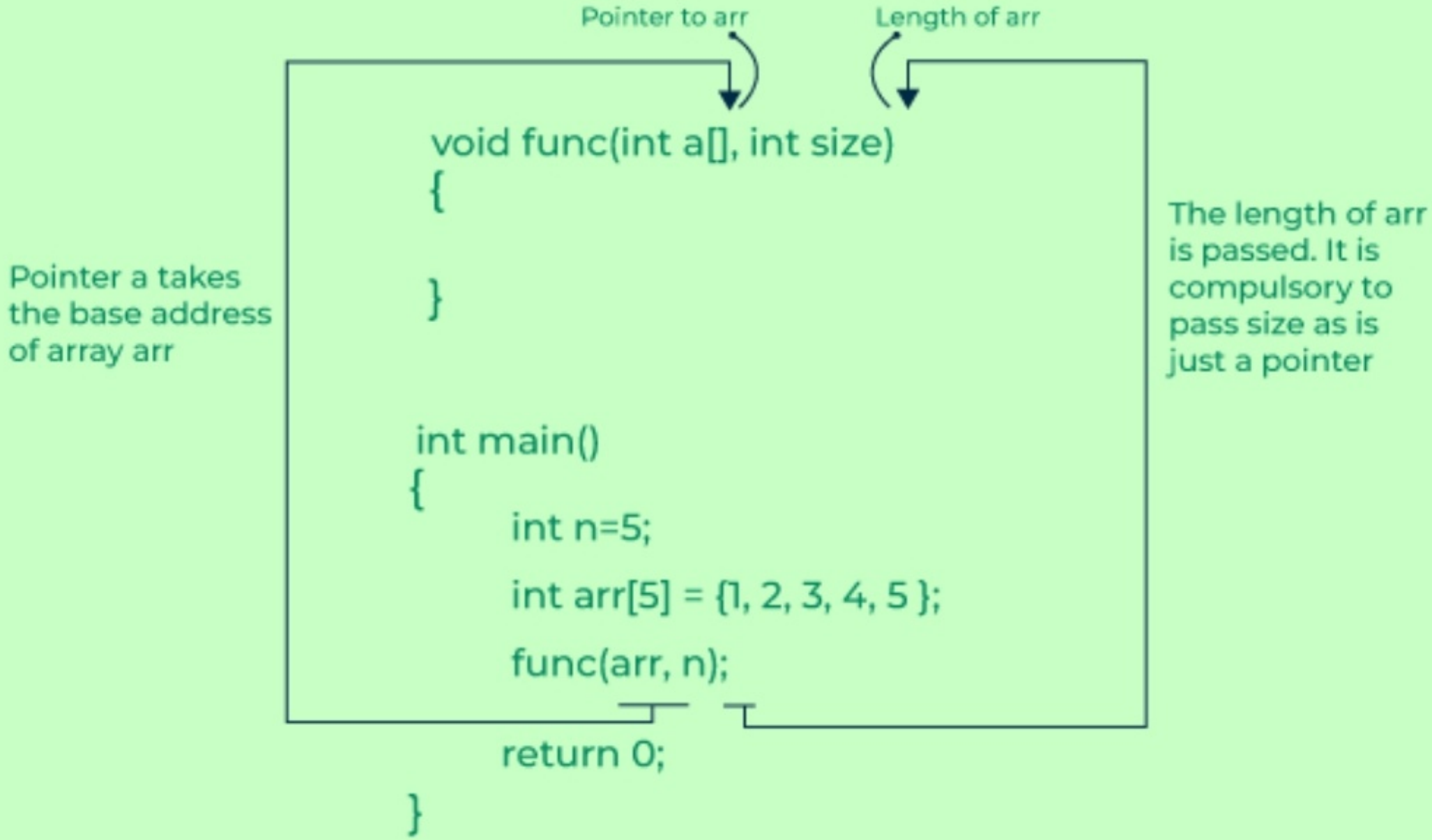
- Deallocates the memory block pointed to by the pointer, making it available for further allocations.
- Syntax:

```c
free(ptr)
```

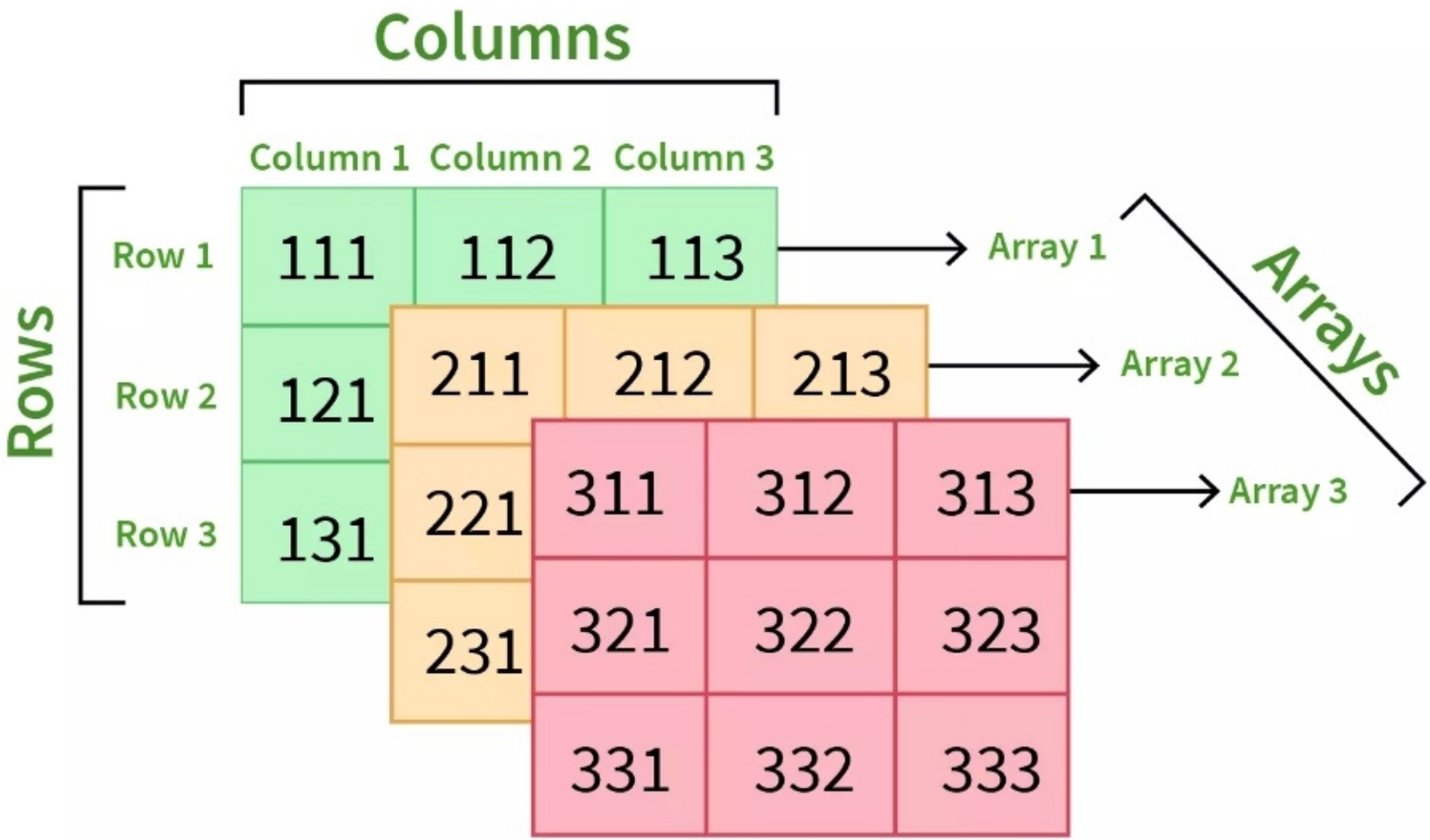
- Example:

```c
free(ptr);
```

Passing Array to Function in C



Nikhilkumar_absolute




BIHAR
DIPLOMA
UNIT-7
[All materials]
t.me/patna7diploma
@nikhilkumar_absolute

Structures

- A structure is a user-defined data type that allows the grouping of variables of different data types under a single name.
- Structures enable the creation of complex data types that can hold various types of data.
- Syntax for defining a structure:

```c
struct tag_name {
    member1_type member1_name;
    member2_type member2_name;
    ...
};
```

- Example:

```c
struct person {
    char name[50];
    int age;
    float height;
};
```

Structure Variables and Arrays

- Structure variables are declared like any other variable.
- Syntax:

```c
struct tag_name variable_name;
```

- Example:

```c
struct person person1;
```

Initialization of Structure Variables

- Structure variables can be initialized at the time of declaration using curly braces.
- Example:

```c
struct person person1 = {"John Doe", 30, 1.75};
```

Arrays of Structures

- Arrays of structures can be created to store multiple instances of the same structure type.
- Syntax:

```c
struct tag_name array_name[size];
```

- Example:

```c
struct person persons[5];
```

Dot (.) Operator

- The dot (`.`) operator is used to access the members of a structure.
- Syntax:

```c
variable_name.member_name
```

- Example:

```c
printf("Name: %s\n", person1.name);
printf("Age: %d\n", person1.age);
```

Unions

- A union is a user-defined data type that allows the storage of different data types in the same memory location.
- Only one member of a union can have a value at a time.
- Syntax for defining a union:

```c
union tag_name {
    member1_type member1_name;
    member2_type member2_name;
    ...
};
```

- Example:

```c
union data {
    int integer;
    float floating;
    char character;
};
```

Union Variables and Arrays

- Union variables are declared like structure variables.
- Syntax:

```c
union tag_name variable_name;
```

- Example:

```c
union data value;
```

- Arrays of unions can be created similar to arrays of structures.
- Syntax:

```c
union tag_name array_name[size];
```

- Example:

```c
union data values[5];
```

	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.

Assigning Value of a Structure to Another Structure

- You can assign the value of one structure variable to another structure variable of the same type using the assignment operator (`=`).
- Example:

```c
struct person {
    char name[50];
    int age;
};

struct person person1 = {"Alice", 25};
struct person person2;

person2 = person1; // Assigns the values of person1 to person2
```

Structure within Structures

- Structures can contain other structures as members, allowing for complex data organization.
- Example:

```c
struct address {
    char street[50];
    char city[50];
};

struct person {
    char name[50];
    int age;
    struct address addr; // Structure within a structure
};
```

Size of a Structure

- The `sizeof()` operator can be used to determine the size of a structure in bytes.
- Example:

```c
printf("Size of person structure: %lu\n", sizeof(struct person));
```

Unions

- A union is similar to a structure but allows storing different data types in the same memory location.
- Only one member of a union can hold a value at any given time, which saves memory.
- Example:

```c
union data {
    int integer;
    float floating;
    char character;
};
```

Size of a Union

- The `sizeof()` operator can also be used to determine the size of a union, which is typically the size of its largest member.
- Example:

```c
printf("Size of data union: %lu\n", sizeof(union data));
```

Difference Between a Structure and a Union

Feature	Structure	Union
Memory Allocation	Allocates memory for all members	Allocates memory for the largest member only
Access	All members can be accessed at once	Only one member can be accessed at a time
Memory Size	Size is the sum of all members' sizes	Size is the size of the largest member
Use Case	Used when you need to store multiple related values	Used when you need to store one of several types of data

Enum Data Type

- An `enum` (enumeration) is a user-defined data type that consists of integral constants, allowing for better readability and maintainability of code.
- Syntax:

```c
enum enum_name {
    constant1,
    constant2,
    ...
};
```

- Example:

```c
enum week { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };
```

- By default, the first enumerator has the value `0`, and the value of each subsequent enumerator is incremented by `1`.

Typedef Declaration

- The `typedef` keyword allows you to create a new name (alias) for an existing data type, improving code readability.
- Syntax:

```c
typedef existing_type new_type_name;
```

- Example:

```c
typedef unsigned long ulong;
ulong bigNumber; // bigNumber is now of type unsigned long
```