YouTube Video Link

KG Coding

Some Other One shot Video Links:

- Complete Java
- Complete HTML & CSS
- Complete JavaScript
- Complete React and Redux
- One shot University Exam Series

http://www.kgcoding.in/

Our YouTube Channels

KG Coding Android App

KG Coding

Knowledge GATE

KG Placement Prep

Sanchit Socket

Installation & Compiler Setup

- What is IDE
- Need of IDE
- Windows Installation
- Mac Installation

What is IDE

1. IDE stands for Integrated Development Environment.
2. Software suite that consolidates basic tools required for software development.
3. Central hub for coding, finding problems, and testing.
4. Designed to improve developer efficiency.

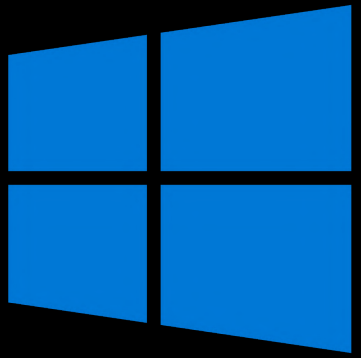
Need of IDE

1. Streamlines development.
2. Increases productivity.
3. Simplifies complex tasks.
4. Offers a unified workspace.
5. IDE Features
    1. Code Autocomplete
    2. Syntax Highlighting
    3. Version Control
    4. Error Checking

MainActivity.kt

```
@Composable
fun MessageCard(msg: Message) {
    Row(modifier = Modifier.padding(all = 8.dp)) {
        Image(
            painter = painterResource(R.drawable.android_studio_logo),
            contentDescription = "Profile Picture",
            modifier = Modifier
                .size(45.dp)
        )

        Spacer(modifier = Modifier.width(8.dp))
        Column (Modifier
            .background(color = Color.White)) {
            Text(text = msg.author, color = Color.Black)
            Spacer(modifier = Modifier.height(1.dp))
            Text(text = msg.body, color = Color.Black)
        }
    }
}
```

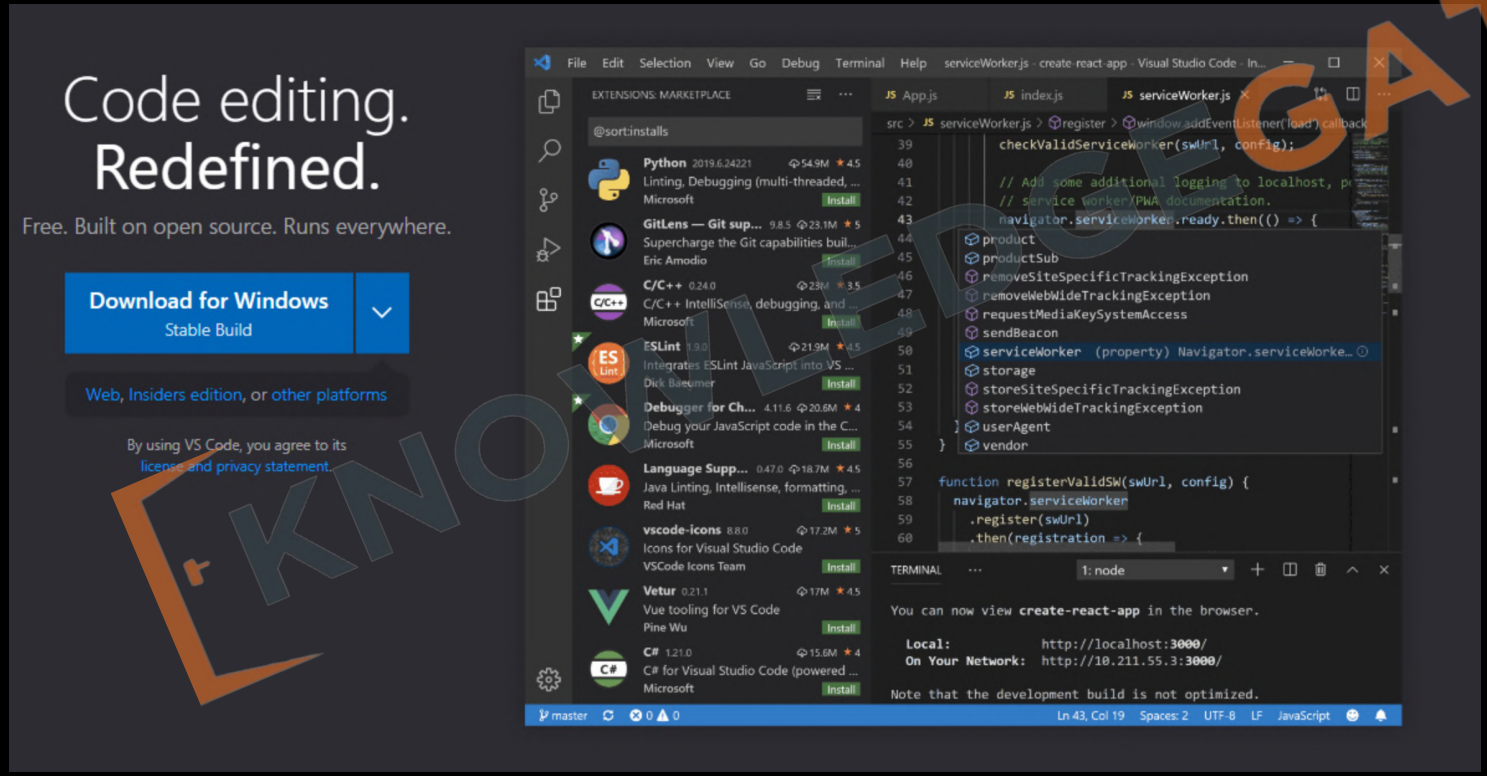
Installation & Setup



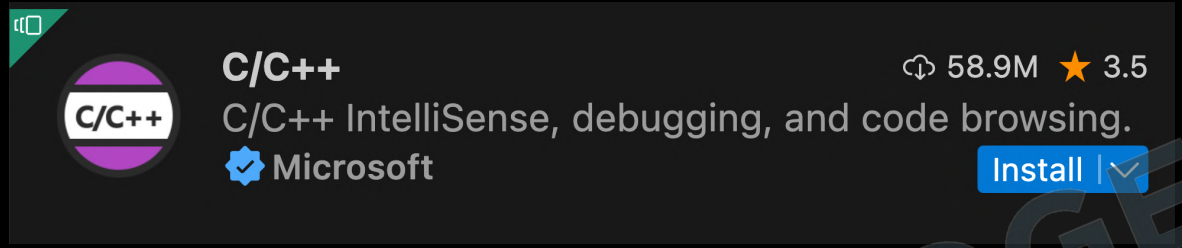
Windows

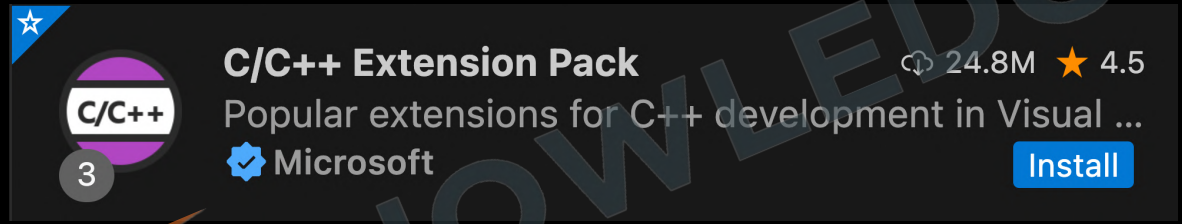
Installing Vs Code

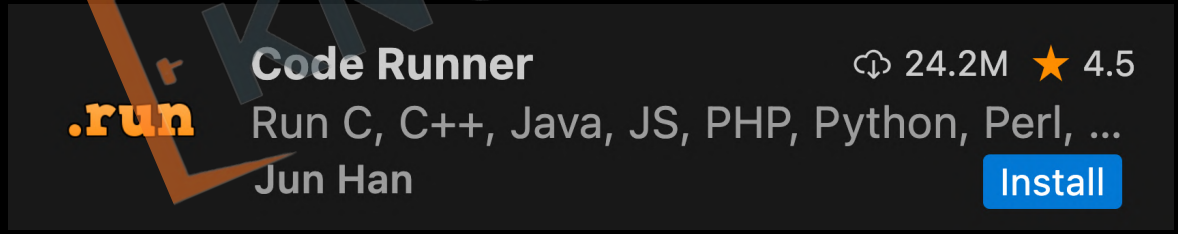
Search VS Code on Google



Vs Code Extensions







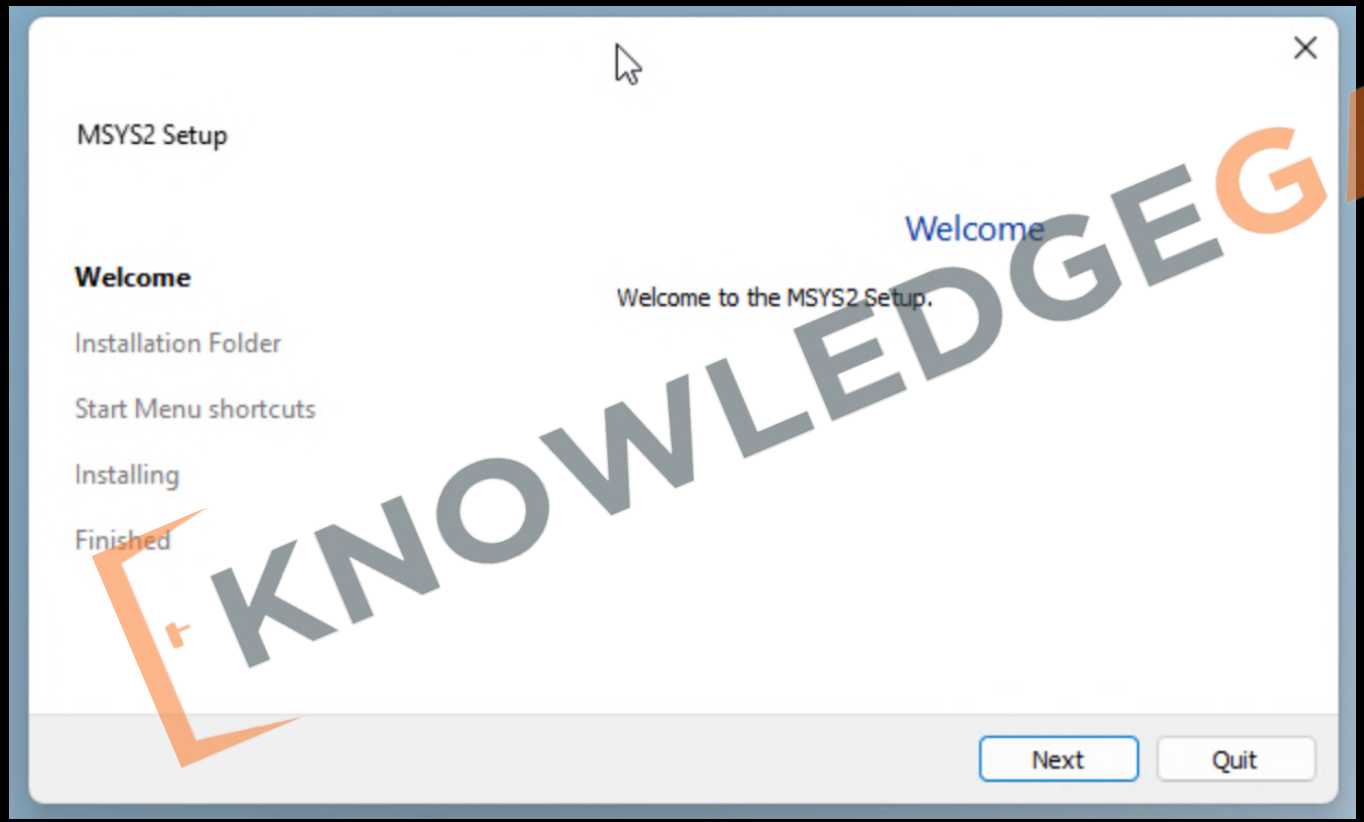
VsCode Settings

1. Line Wrap
2. Tab Size from 4 to 2

Downloading msys-2

https://aka.ms/Install_MinGW

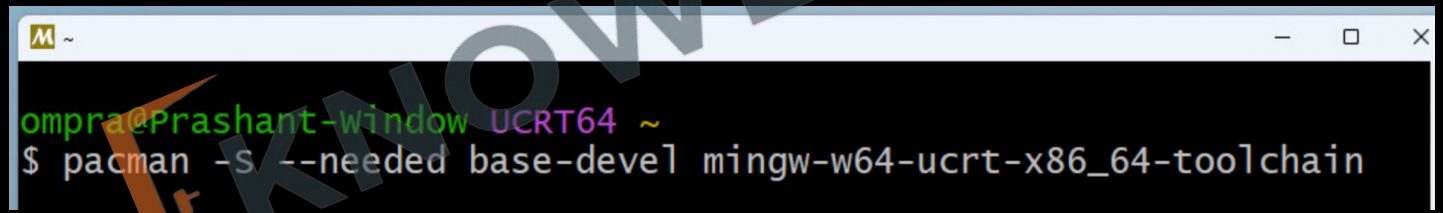
Installing msys-2



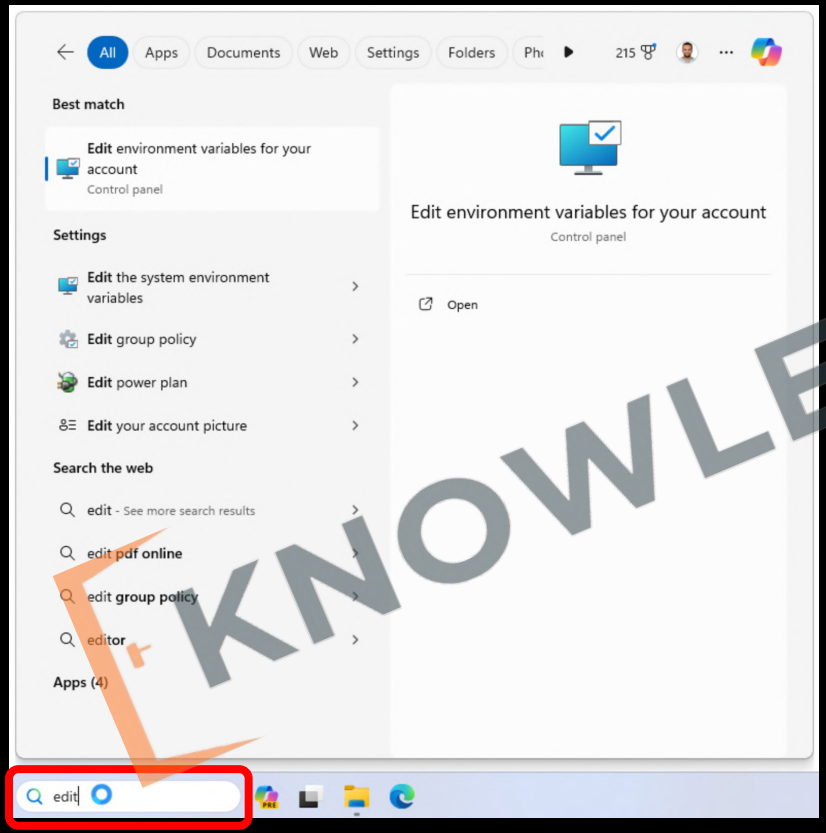
Installing MinGW

4. In this terminal, install the MinGW-w64 toolchain by running the following command:

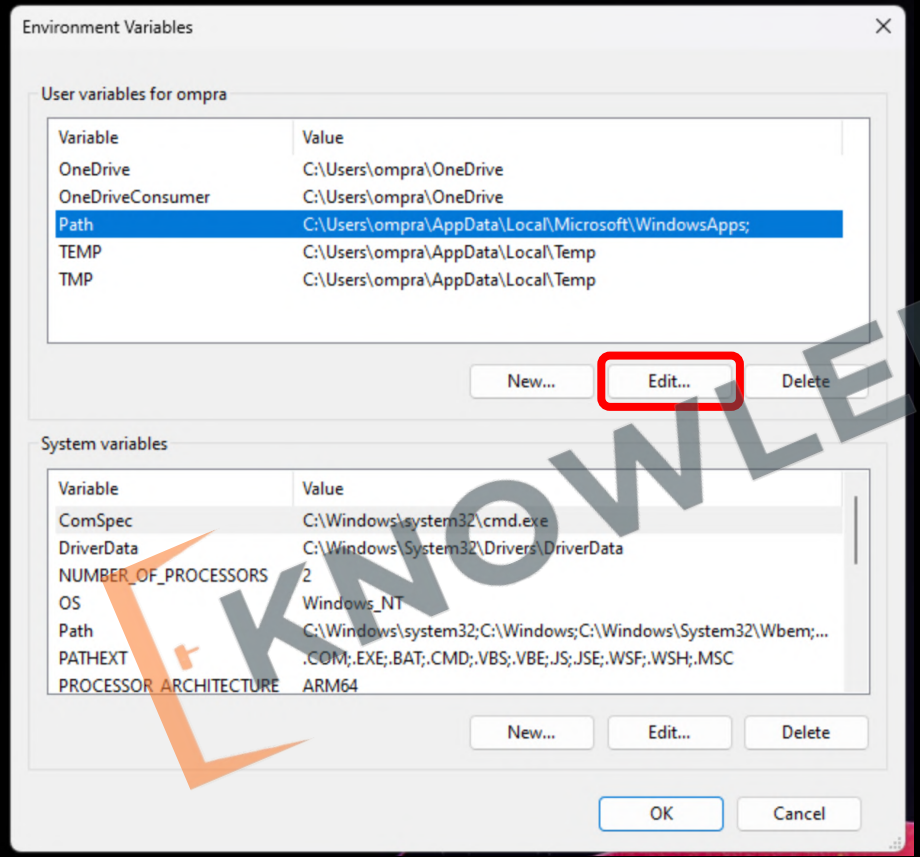
```
pacman -S --needed base-devel mingw-w64-ucrt-x86_64-toolchain
```



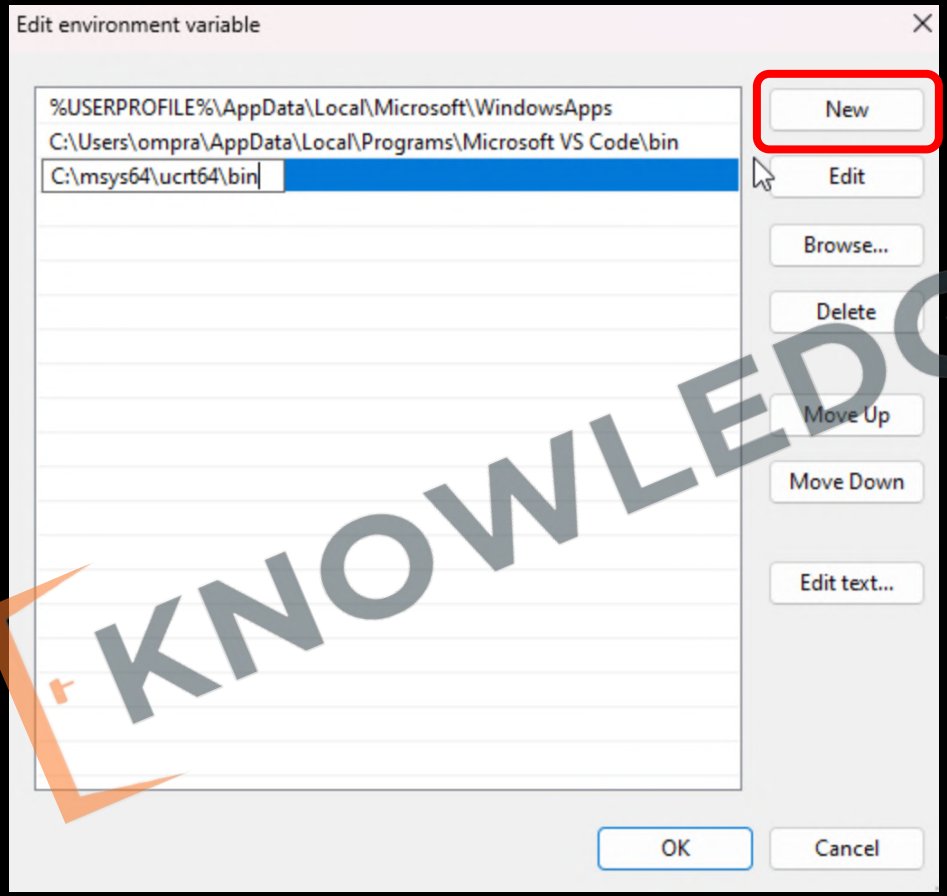
Adding Path



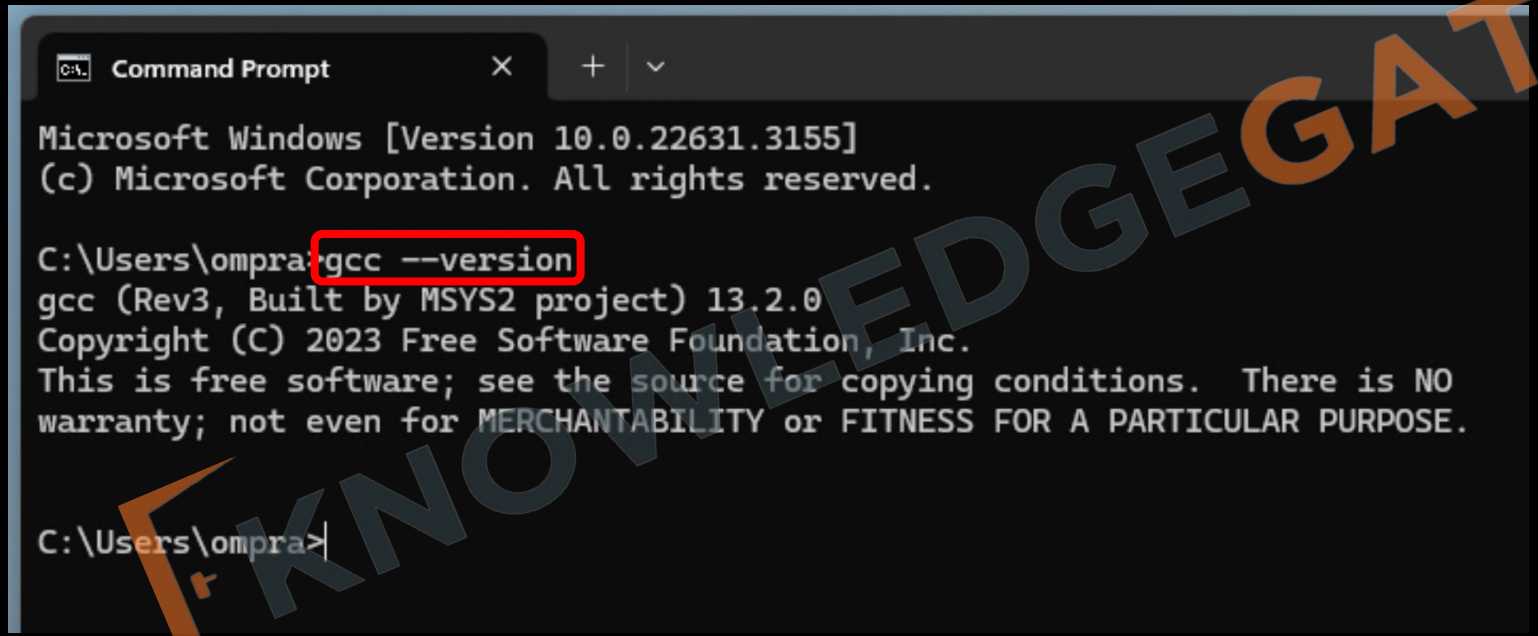
Adding Path



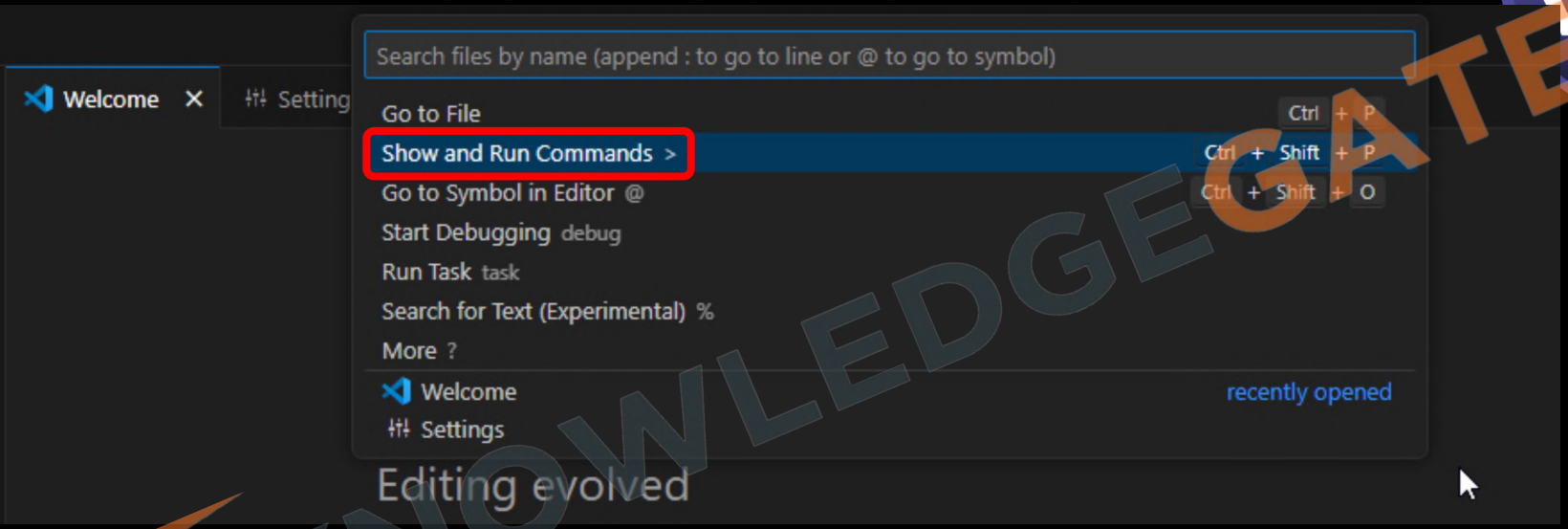
Adding Path



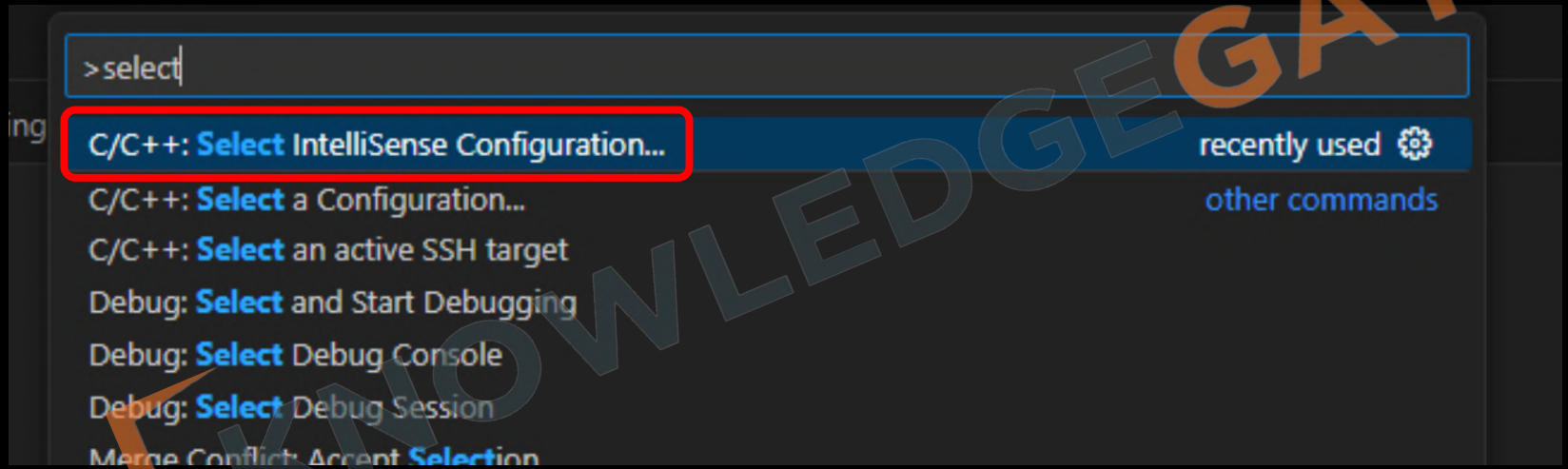
Verifying Path



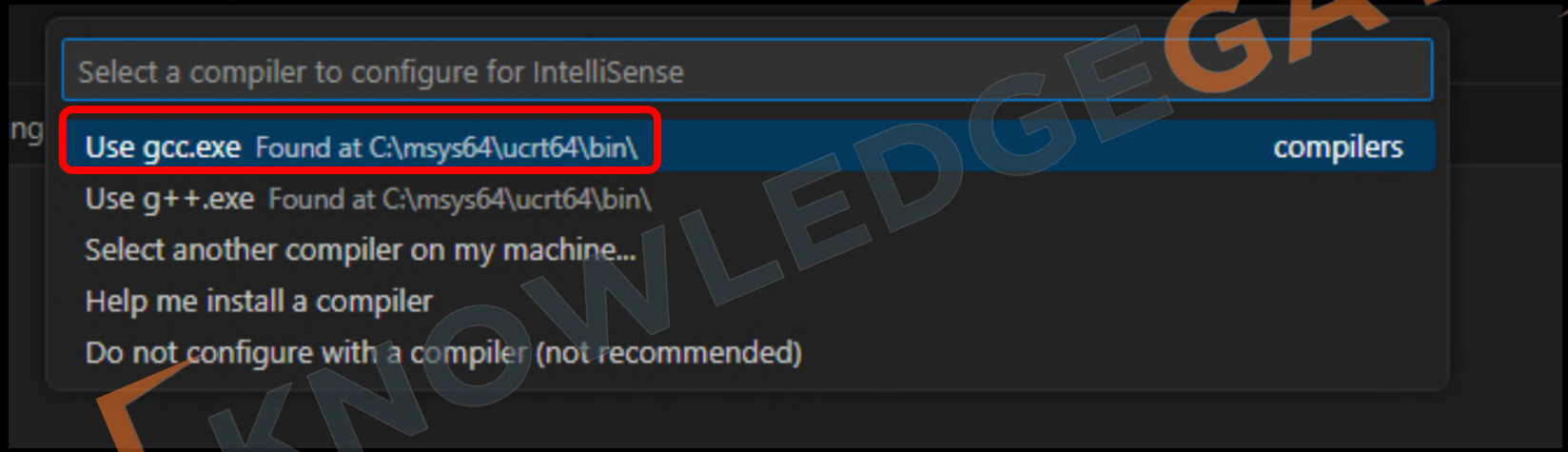
Configuring Path in VS Code



Configuring Path in VS Code



Configuring Path in VS Code

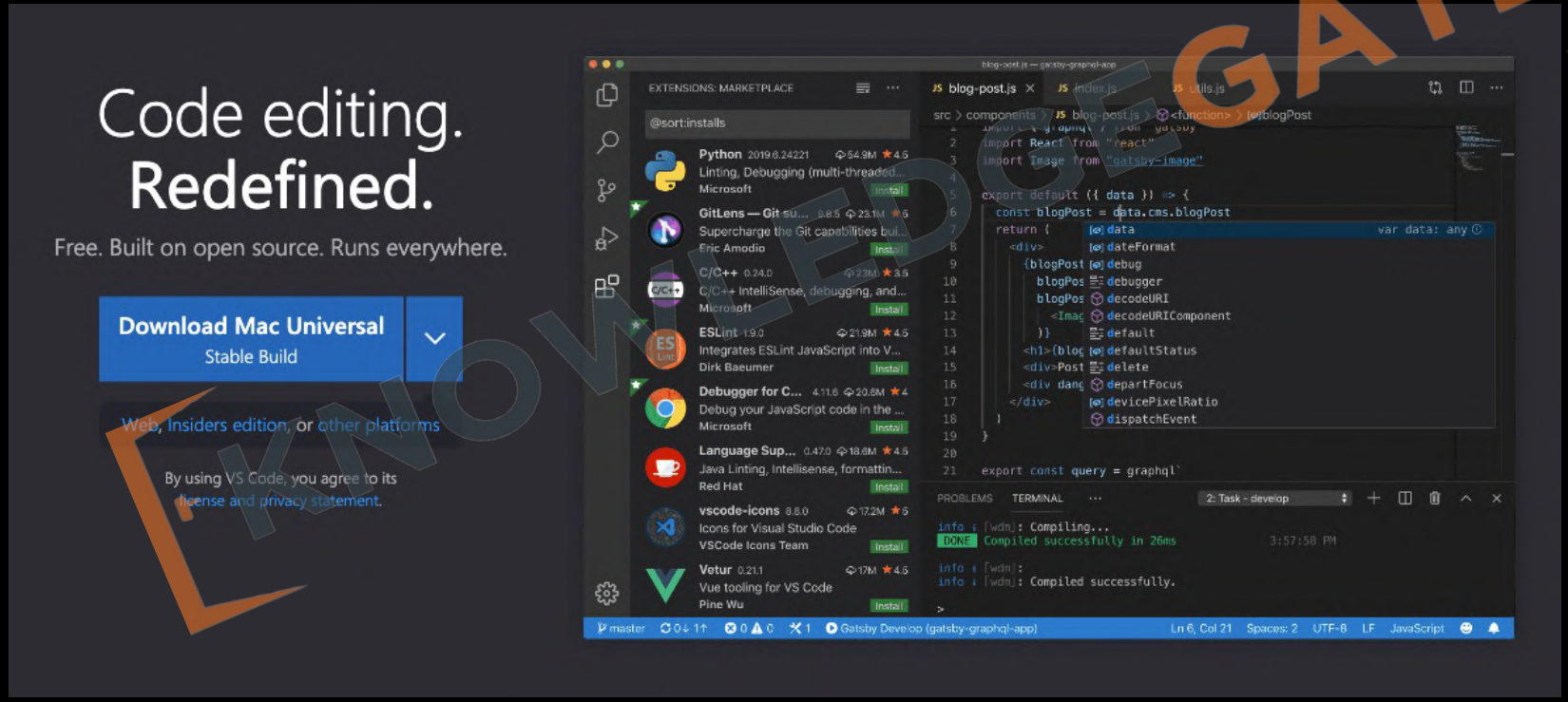


Installation & Setup

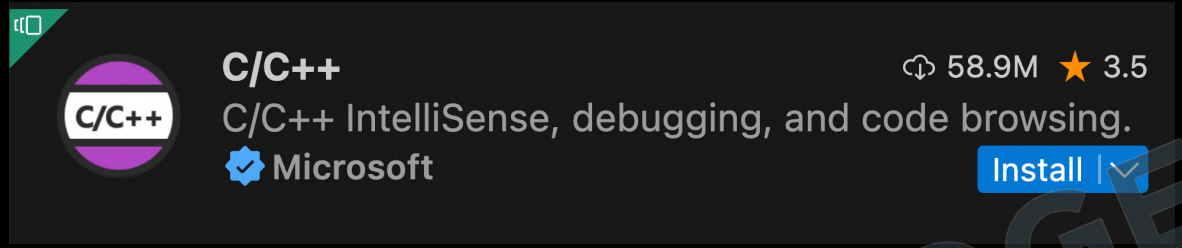
mac OS

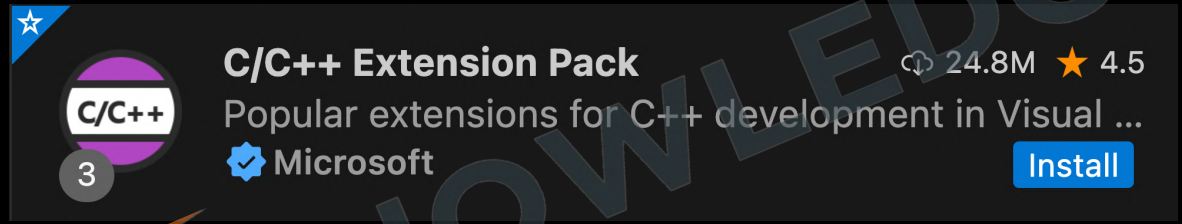
Installing Vs Code

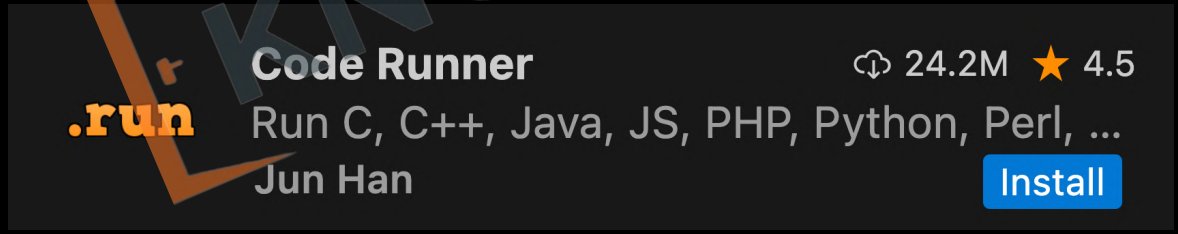
Search VS Code on Google



Vs Code Extensions







VsCode Settings

1. Line Wrap
2. Tab Size from 4 to 2

Installing Clang

Ensure Clang is installed

Clang might already be installed on your Mac. To verify that it is, open a macOS Terminal window and enter the following command:

```
clang --version
```

If Clang isn't installed, enter the following command to install the command line developer tools, which include Clang:

```
xcode-select --install
```

Running Hello World

test.c > main()

```c
#include <stdio.h>
int main()
{
    // printf() displays the string inside quotation
    printf("Hello, World!");
    return 0;
}
```

KG Coding

Some Other One shot Video Links:

- Complete Java
- Complete HTML & CSS
- Complete JavaScript
- Complete React and Redux
- One shot University Exam Series

http://www.kgcoding.in/

Our YouTube Channels

KG Coding Android App

KG Coding

Knowledge GATE

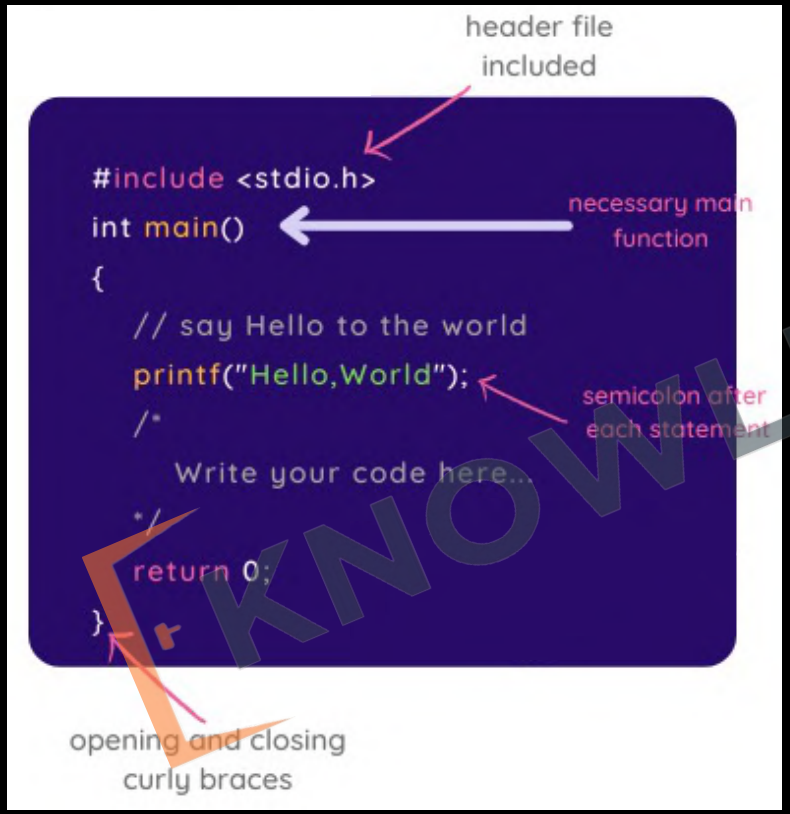
KG Placement Prep

Sanchit Socket

1. First C Program

1. Basic Program Structure
2. Showing output
3. Importance of main method
4. File Extension
5. Comments
6. Coding using Command line
7. What is a Programming Language
8. What is an Algorithm
9. What is Syntax
10. What is a Compiler

1.1 Basic Program Structure

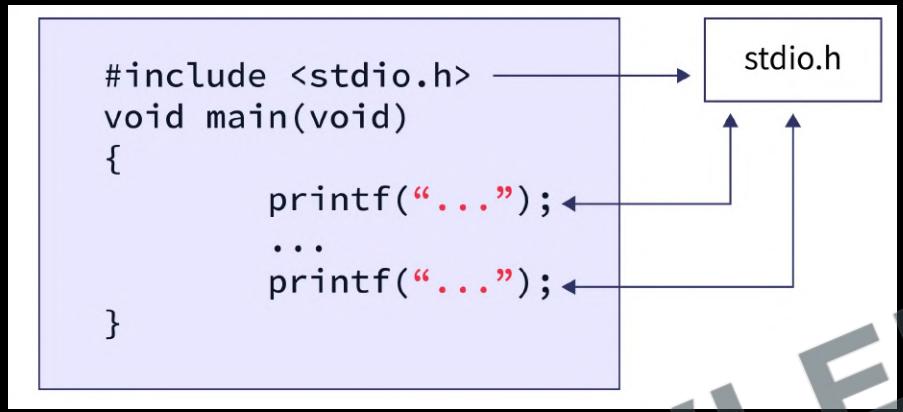


1.2 Showing output

- The printf function is used for output. It allows you to display text and variables to the console. Syntax: printf("format string", variable1, variable2, ...);
- Displaying Text: To print text, enclose the message in double quotes. printf("Hello, World!");
- New Line: Use \n within the string to move to a new line. printf("Hello\nWorld");
- Inserting Values: Use format specifiers like:
  - %d or %i : for integers
  - %c : for characters
  - %f : for decimal numbers

```c
#include <stdio.h>

int main() {
    // Printing integer 10 using %d
    printf("Integer: %d\n", 10);

    // Printing character 'A' using %c
    printf("Character: %c\n", 'A');

    // Printing floating-point number 3.14 using %f
    printf("Floating-point: %f\n", 3.14);

    return 0;
}
```

1.3 Importance of the main method



- Entry Point: It's the entry point of a C program, where the execution starts. When a C program is executed, the control starts from the main function.
- Required: Every executable C program must have a main function.
- Return Type: Typically returns int, indicating program success (0) or error (non-zero) to the operating system.
- Fixed Name: The name main is recognized by C compilers as the program's starting point.

1.4 File Extension

.c

- Contain Executable Code
- Compiled: Transformed into executable programs.
- Often hosts the main

.h

- Contain Declarations for code to be shared.
- Improve modularity and manageability in projects.
- Facilitate code use across multiple .c files without duplication.

test.c

```c
#include <stdio.h>

int main()
{
    // Printing integer 10 using %d
    printf("Integer: %d\n", 10);
    return 0;
}
```

1.5 Comments

1. Used to add notes in C code
2. Not considered as part of code
3. Helpful for code organization
4. Syntax:
    1. Single Line: //
    2. Multi Line: /* */

```c
#include <stdio.h>

int main() {
    // This is a single-line comment explaining the next line
    printf("Hello, world!\n");

    /* This is a multi-line comment
    I can keep talking out it
    in multiple lines */

    printf("This line of code is great");

    return 0;
}
```

1.6 Coding using Command line

Coding

- Write code using a text editor
- Save the file with a .c extension

Compiling and running

- Use gcc file –o outputname command to compile.
- Run the output file using:
  - On Windows: .\outputname
  - On macOS/Linux: ./outputname

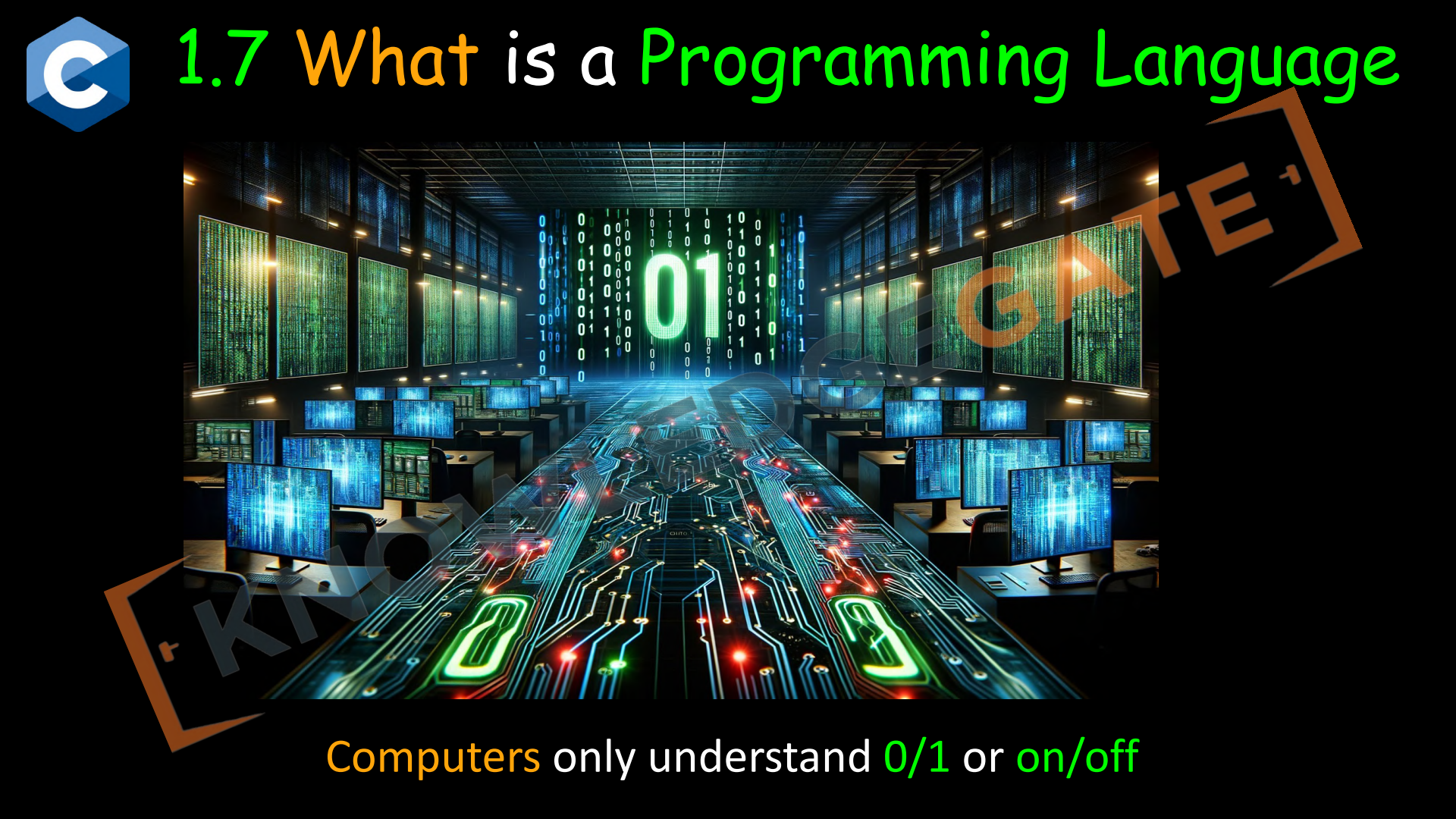
text.c — Edited

```
#include <stdio.h>

int main() {
    printf("Hello, world!\n");
    return 0;
}
```

```
prashantjain@Mac-mini c test % ls
test.c
prashantjain@Mac-mini c test % gcc test.c -o hello.out
prashantjain@Mac-mini c test % ./hello.out
Hello World!%
prashantjain@Mac-mini c test %
```

1.7 What is a Programming Language

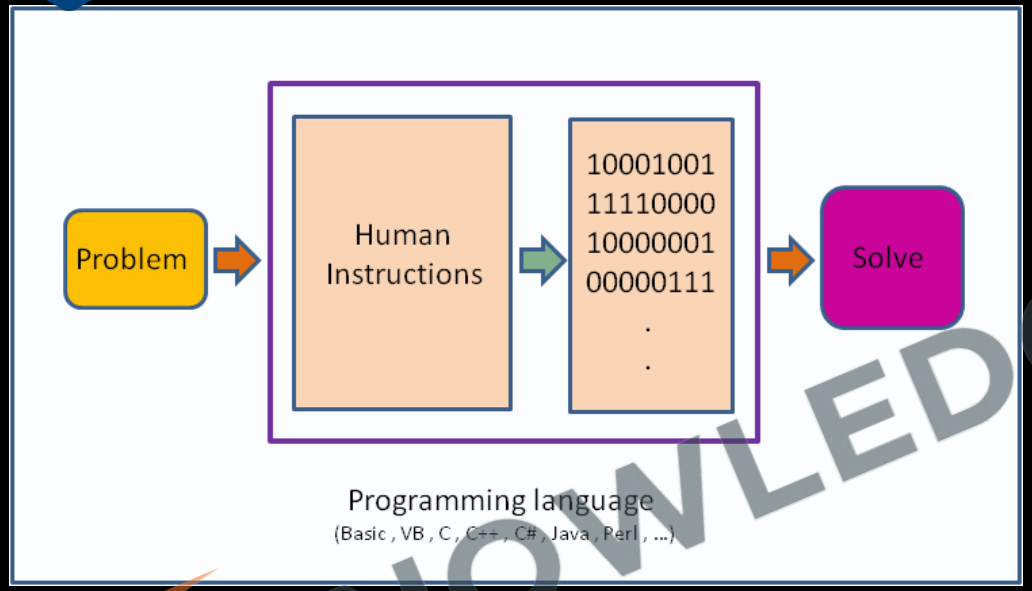
Humans use natural language (like Hindi/English) to communicate



1.7 What is a Programming Language

Computers only understand 0/1 or on/off

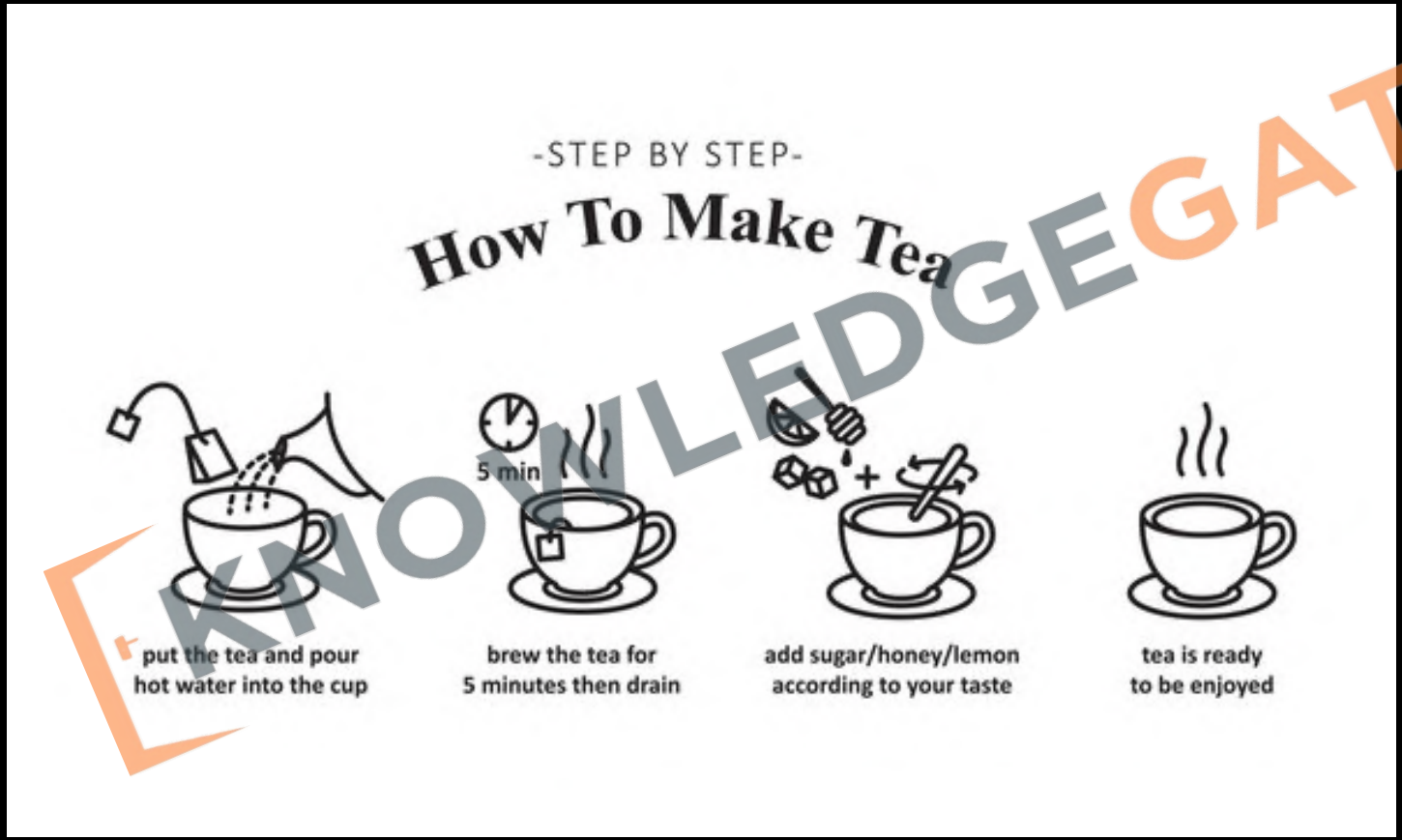
1.7 What is a Programming Language



- Giving instructions to a computer
- **Instructions**: Tells computer what to do.
- These instructions are called code.
- Human instructions are given in High level languages.

Compiler converts high level languages to low level languages or machine code.

1.8 What is an Algorithm



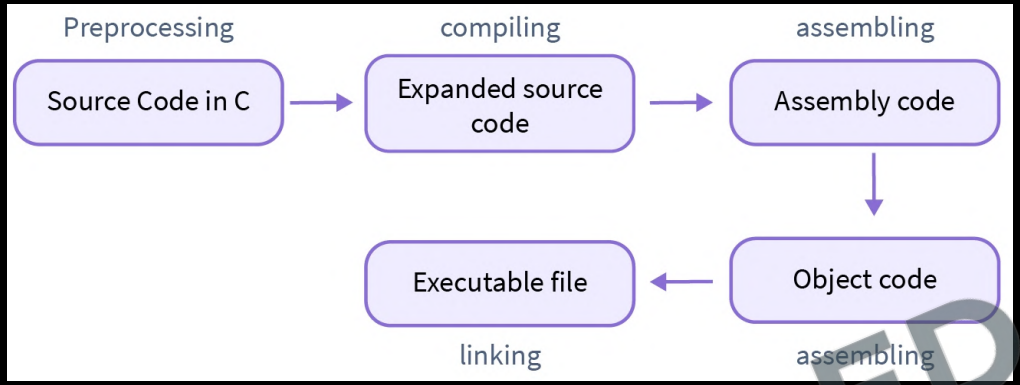
1.8 What is an Algorithm

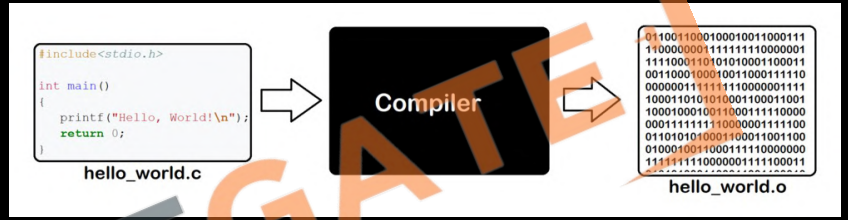
An algorithm is a step-by-step procedure for solving a problem or performing a task.

1.9 What is Syntax

- Structure of words in a sentence.
- Rules of the language.
- For programming exact syntax must be followed.

1.10 What is a Compiler





- Pre-processing: Processes directives (like #include and #define) before compilation, modifying the source code according to these instructions.
- Compiling: Transforms source code written in a high-level language (like C) into assembly language.
- Assembling: Converts assembly language into machine code, generating object files.
- Linking: Combines multiple object files into a single executable or library, resolving references between them.

Revision

1. Basic Program Structure
2. Showing output
3. Importance of main method
4. File Extension
5. Comments
6. Coding using Command line
7. What is a Programming Language
8. What is an Algorithm
9. What is Syntax
10. What is a Compiler

KG Coding

Some Other One shot Video Links:

- Complete Java
- Complete HTML & CSS
- Complete JavaScript
- Complete React and Redux
- One shot University Exam Series

http://www.kgcoding.in/

Our YouTube Channels

KG Coding Android App

KG Coding

Knowledge GATE

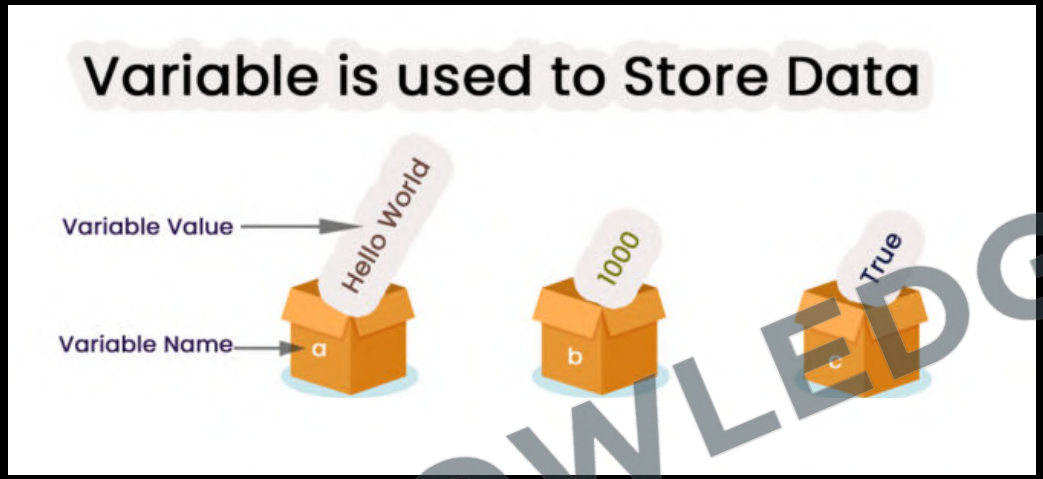
KG Placement Prep

Sanchit Socket

2 Variables, Data Types & Input/Output

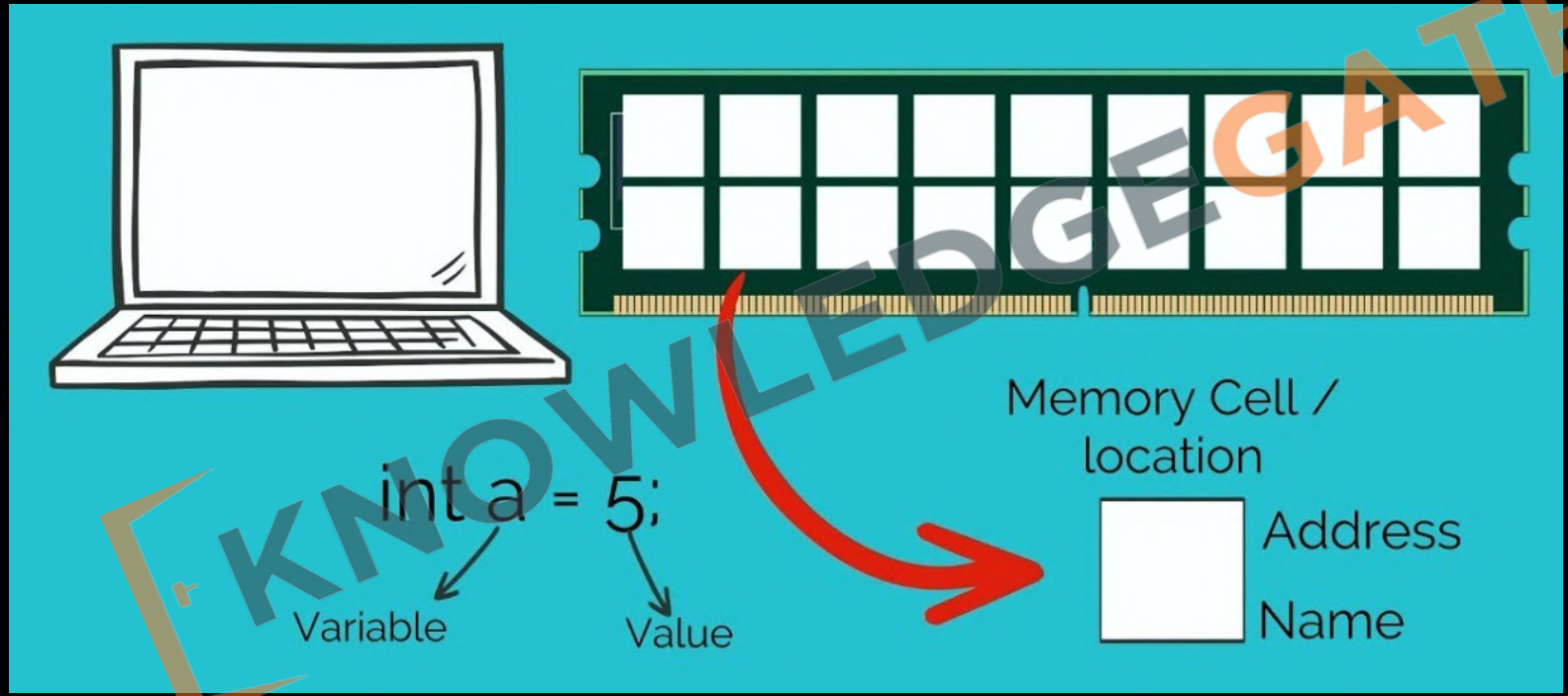
1. Variables
2. Data Types
3. Naming Conventions
4. Literals
5. Constants
6. Keywords
7. Escape Sequences
8. User Input using Scanf
9. Sum of Two Numbers

2.1 What are Variables?

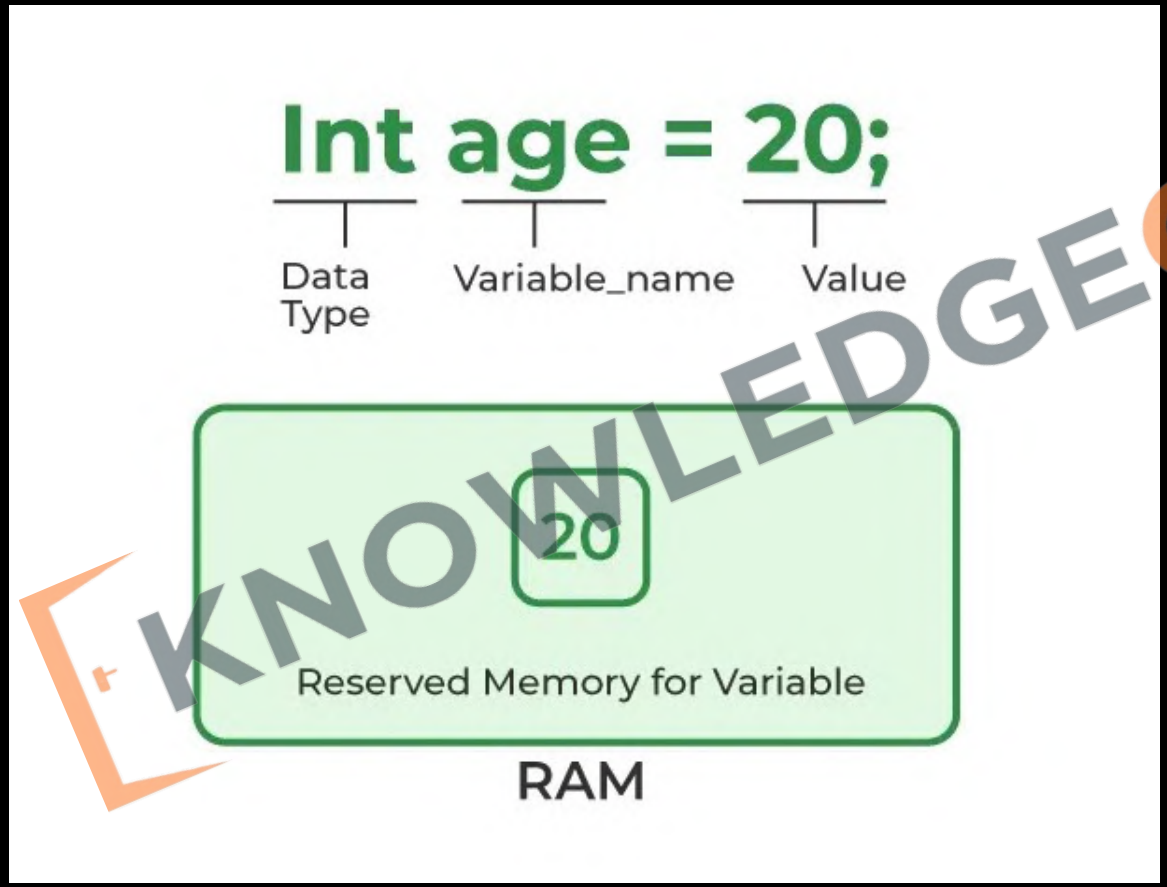


Variables are like containers used for storing data values.

2.1 Memory Allocation



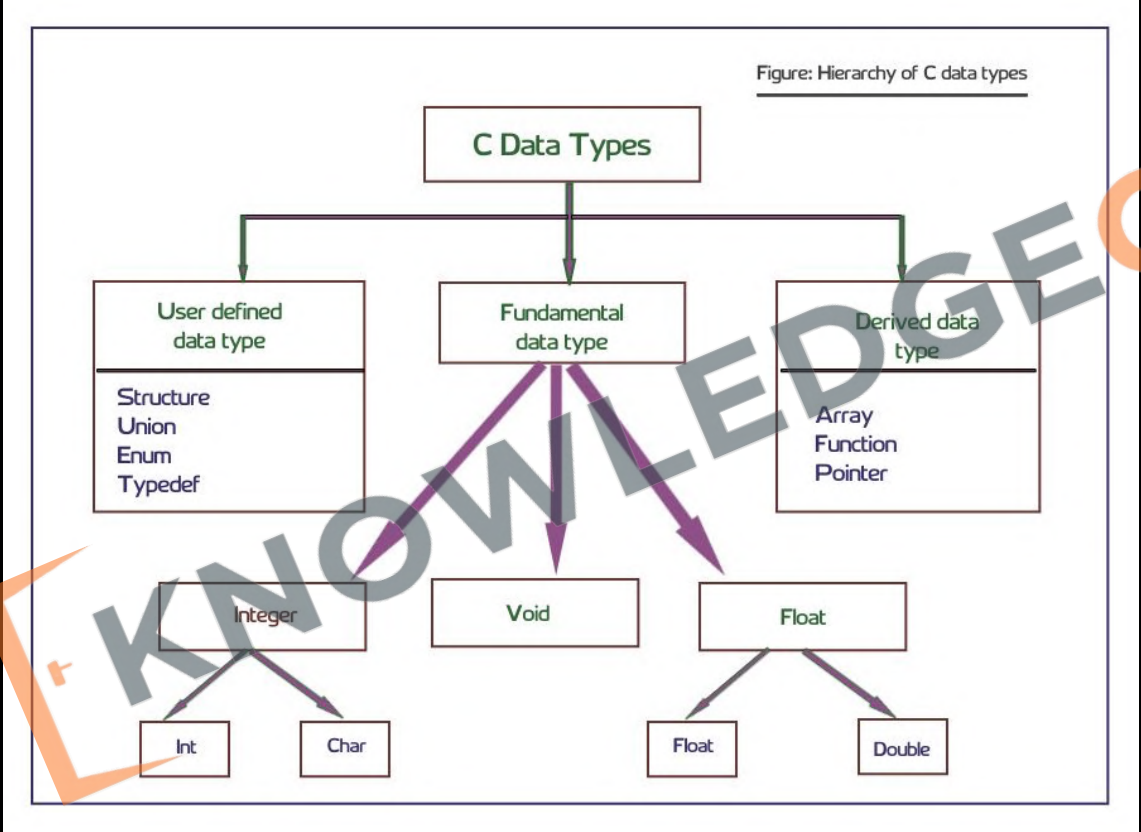
2.1 Variable Declaration



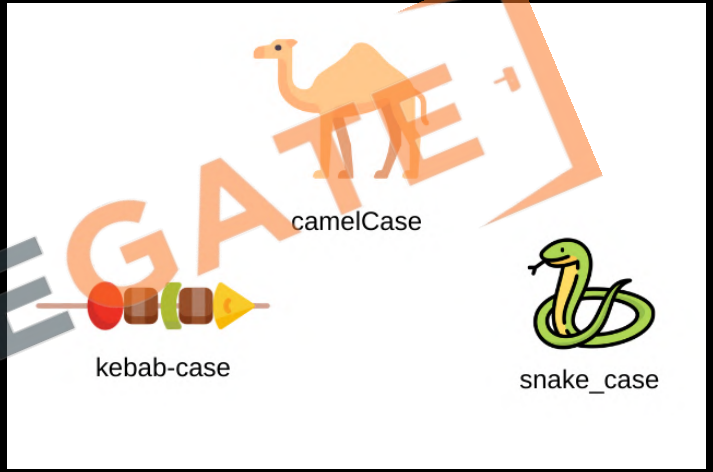
2.2 Data Types

C Basic Data Types	32-bit CPU		64-bit CPU	
	Size (bytes)	Range	Size (bytes)	Range
char	1	-128 to 127	1	-128 to 127
short	2	-32,768 to 32,767	2	-32,768 to 32,767
int	4	-2,147,483,648 to 2,147,483,647	4	-2,147,483,648 to 2,147,483,647
long	4	-2,147,483,648 to 2,147,483,647	8	-9,223,372,036,854,775,808-9,223,372,036,854,775,807
long long	8	9,223,372,036,854,775,808-9,223,372,036,854,775,807	8	9,223,372,036,854,775,808-9,223,372,036,854,775,807
float	4	3.4E +/- 38	4	3.4E +/- 38
double	8	1.7E +/- 308	8	1.7E +/- 308

2.2 Data Types



2.3 Naming Conventions



camelCase

- Start with a lowercase letter. Capitalize the first letter of each subsequent word.
- Example: myVariableName

snake_case

- Start with an lowercase letter. Separate words with underscore
- Example: my_variable_name

Kebab-case

- All lowercase letters. Separate words with hyphens. Example: my-variable-name

Keep a Good and Short Name

- Choose names that are descriptive but not too long. It should make it easy to understand the variable's purpose.
- Example: age, first_name, is_married

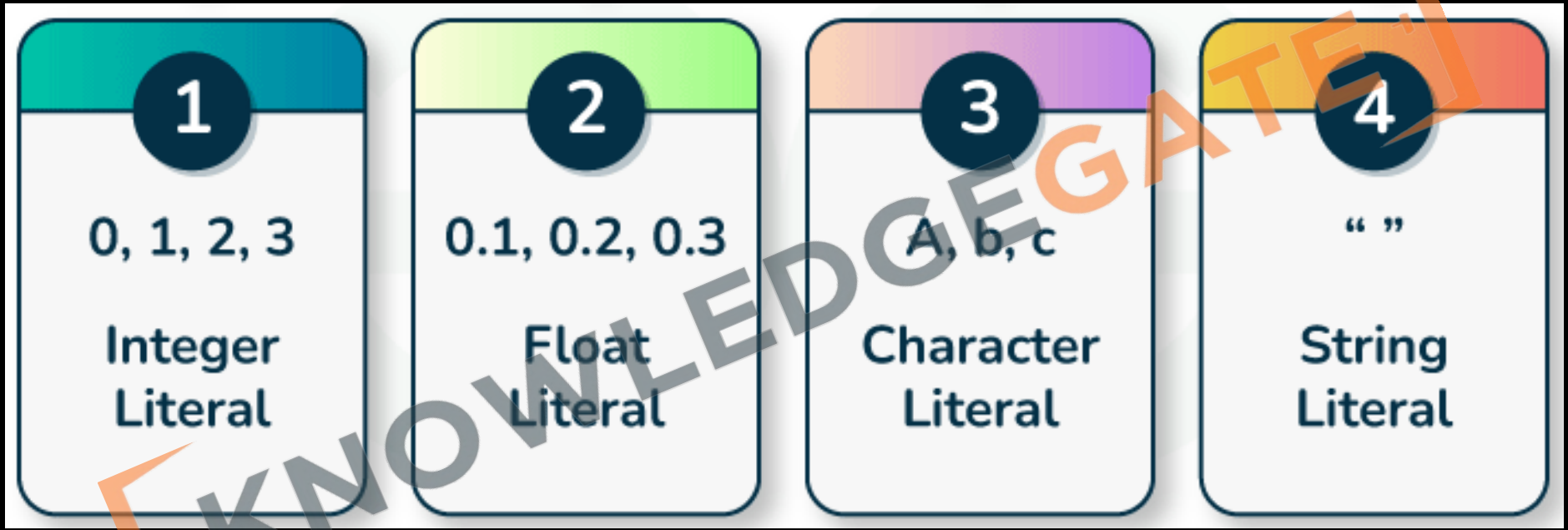
2.3 C Identifier Rules

1. The only allowed characters for identifiers are all alphanumeric characters([A-Z],[a-z],[0-9]), and '_' (underscore).
2. Can't use keywords or reserved words
3. Identifiers should not start with digits([0-9]).
4. C identifiers are case-sensitive.
5. There is no limit on the length of the identifier but it is advisable to use an optimum length of 4 – 15 letters only.

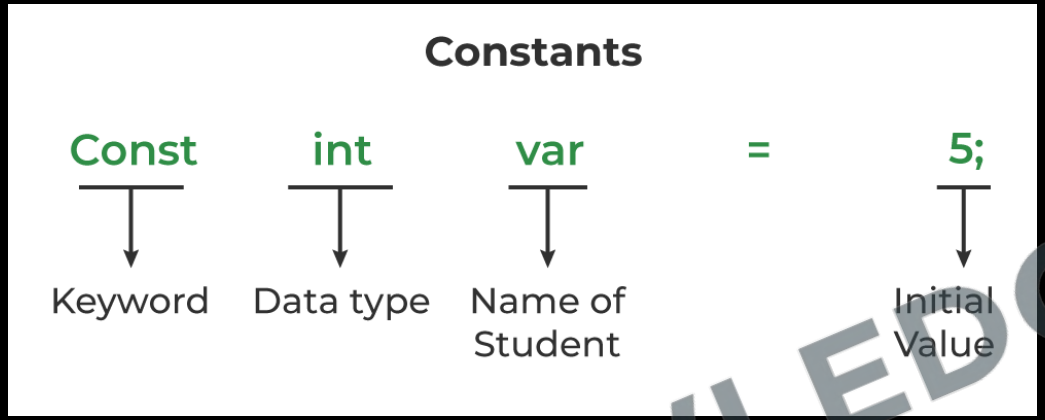
1abc	(invalid)
num	(valid)
First second	(invalid)
_1a	(valid)

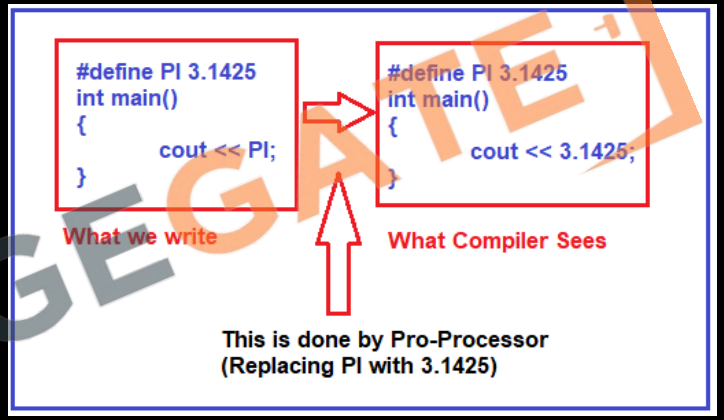
2	(invalid)
a#c	(invalid)
First-second	(invalid)

2.4 Literals



2.5 Constants





1. Fixed values in C code that do not change during execution.
2. Constants are defined using #define or const
   (e.g., #define PI 3.1425).
3. Enhances code readability and eases modifications.
4. Constants are immutable.

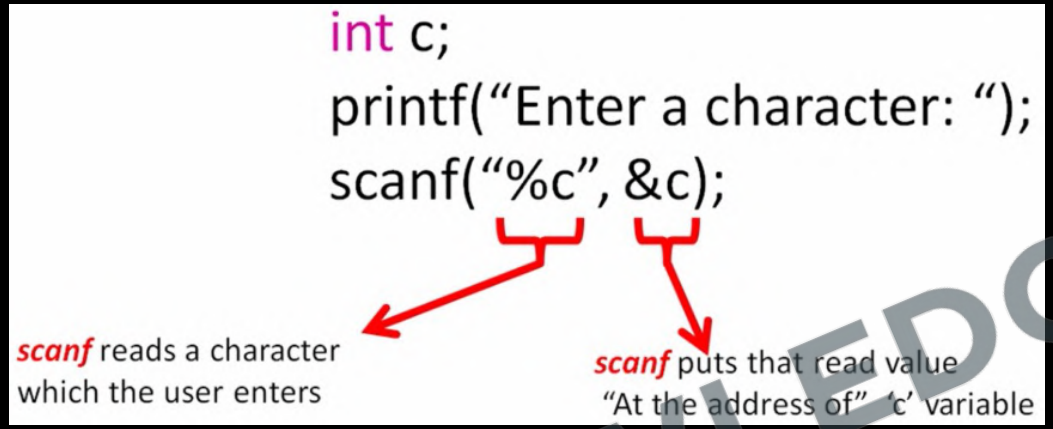
2.6 Keywords

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

2.7 Escape Sequences

Escape Sequence	Description
\t	Insert a tab in the text at this point.
\b	Insert a backspace in the text at this point.
\n	Insert a newline in the text at this point.
\'	Insert a single quote character in the text at this point.
\"	Insert a double quote character in the text at this point.
\\	Insert a backslash character in the text at this point.

2.8 User Input using scanf



1. scanf is used for reading formatted input.
2. Syntax: scanf("format specifier", &variable);
3. Use format specifiers (e.g., %d for integers) to define the input type.
4. Address Operator: Prefix variables with &, except for arrays and strings.

2.9 Sum of Two Numbers

```c
#include <stdio.h>

int main() {
    int num1, num2, sum;

    // Prompt the user for two numbers
    printf("Enter first number: ");
    scanf("%d", &num1);
    printf("Enter second number: ");
    scanf("%d", &num2);

    // Calculate the sum
    sum = num1 + num2;

    // Display the result
    printf("The sum of %d and %d is: %d\n", num1, num2, sum);
    return 0;
}
```

Revision

1. Variables
2. Data Types
3. Naming Conventions
4. Literals
5. Constants
6. Keywords
7. Escape Sequences
8. User Input using Scanf
9. Sum of Two Numbers

Tasks:

1. Show the following patterns just using print statements:

2. Show the following patterns using single print statement:

CHALLENGE

3. Create a program to input name of the person and respond with ”Welcome NAME to KG Coding”
4. Create a program to declare two integer variables, assign them values, and display their values.
5. Create a program that declares one variable of each of the fundamental data types (int, float, double, char) and prints their size using sizeof() operator.
6. Define variables for storing a user's first name, last name, and age using appropriate naming conventions and then display them.
7. Create a program to print the area of a square by inputting its side length
8. Create a program to define a constant for the mathematical value pi (3.14159) and use it to calculate and print the circumference of a circle with a radius input from user.
9. Create a program to print the area of a circle by inputting its radius.
10. Create a program to swap two numbers.

KG Coding

Some Other One shot Video Links:

- Complete Java
- Complete HTML & CSS
- Complete JavaScript
- Complete React and Redux
- One shot University Exam Series

http://www.kgcoding.in/

Our YouTube Channels

KG Coding Android App

KG Coding

Knowledge GATE

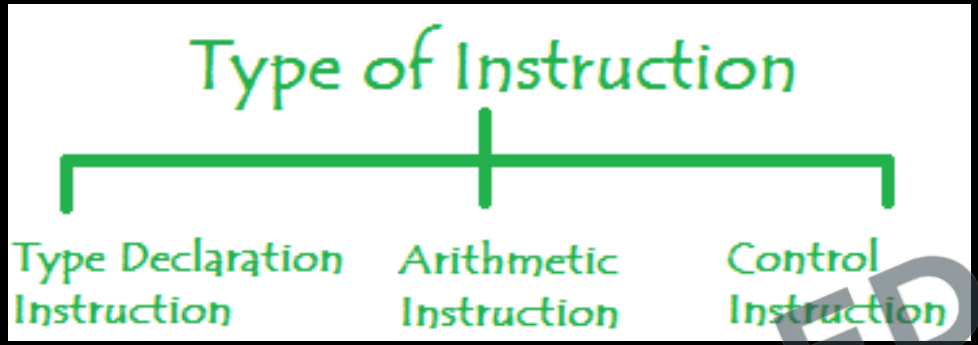
KG Placement Prep

Sanchit Socket

3. Instructions, Expressions & Operators

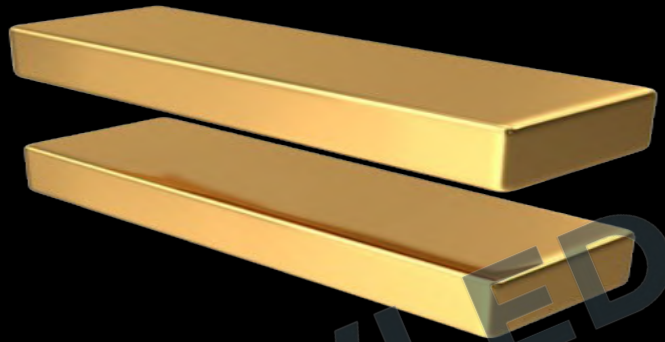
1. C Instructions
2. Type Declaration Instruction
3. Arithmetic Operators
4. Arithmetic Instruction
5. Integer and Float Conversions
6. Type Conversions
7. Hierarchy of Operations
8. Associativity of Operations
9. Shorthand Operators
10. Unary Operators
11. Control Instructions

3.1 C Instructions



1. Type Declaration Instruction: To declare the type of variables used in a C program.
2. Arithmetic Instruction: To perform arithmetic operations between constants and variables.
3. Control Instruction: To control the sequence of execution of various statements in a C program.

3.2 Assignment Operator



Assigns the right-hand operand's value to the left-hand operand.

Example: int a = 5;.

3.2 Type Declaration Instruction

```c
// Valid
int age;
int i = 10, j = 25;
float temperature = 98.6;
float a = 1.5, b = 1.99 + 2.4 * 1.44;
float p = 5, q = p * 6;
char name[50];
int x, y, z = 5;
```

```c
// Invalid
score = 100; // Data type is missing
int distance = 3.14; // Assigning a float to an int
char grade: 'A'; // : should be replaced by =
float return; // 'return' is a keyword
number num; // 'number' is not a valid C data type
float b = a + 3.1, a = 1.5; // using a before declaring it
int x = y = z = 10; // using y before defining it
```


1. Define variable and function data types, guiding memory allocation.
2. Syntax: data_type variable_name; e.g., int age;.
3. Common Types: Include int, float, char, double.
4. Allows immediate value assignment, e.g., int count = 10;.
5. Scope: Dictates variable visibility, with local scope inside functions and global scope outside.

3.3 Arithmetic Operators

Operators	Meaning	Example	Result
+	Addition	4+2	6
-	Subtraction	4-2	2
*	Multiplication	4*2	8
/	Division	4/2	2
%	Modulus operator to get remainder in integer division	5%2	1

3.4 Arithmetic Instruction

Variable on the left of =, and right side can have a combination of variables, arithmetic operators & constants.

```
int ad;
float kot, delta, alpha, beta, gamma;
ad = 3200;
kot = 0.0056;
delta = alpha * beta / gamma + 3.2 * 2/5;
```

Types

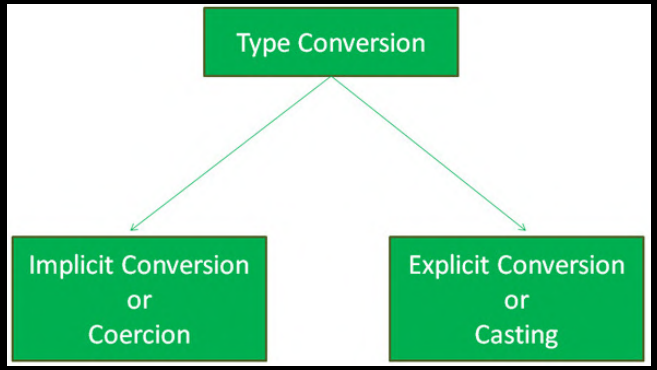
1. Integer Mode
2. Real Mode
3. Mixed Mode

3.5 Integer and Float Conversions

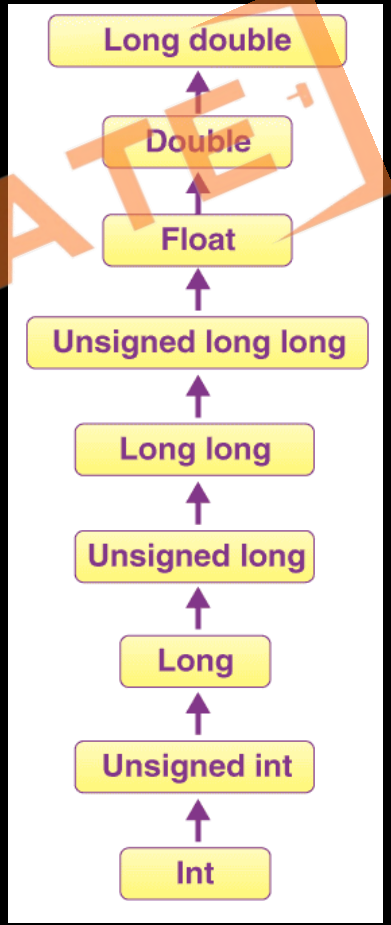
Operation	Result	Operation	Result
5 / 2	2	2 / 5	0
5.0 / 2	2.5	2.0 / 5	0.4
5 / 2.0	2.5	2 / 5.0	0.4
5.0 / 2.0	2.5	2.0 / 5.0	0.4

1. Arithmetic operation between an integer and integer always yields an integer.
2. Arithmetic operation between a real and real always yields a real.
3. Arithmetic operation between an integer and a real always yields a real result. Integer is first promoted to real and then the operation is performed.

3.6 Type Conversions



1. Implicit Conversion: C automatically changes one data type to another when needed (e.g., int to float in operations).
2. Promotion: Smaller types (like char) are automatically promoted to int in expressions.
3. Assignment Conversion: When assigning values, the type is converted to match the variable's type.
4. Casting: Use (type_name) to explicitly convert a value to a different type (e.g., (float)var).



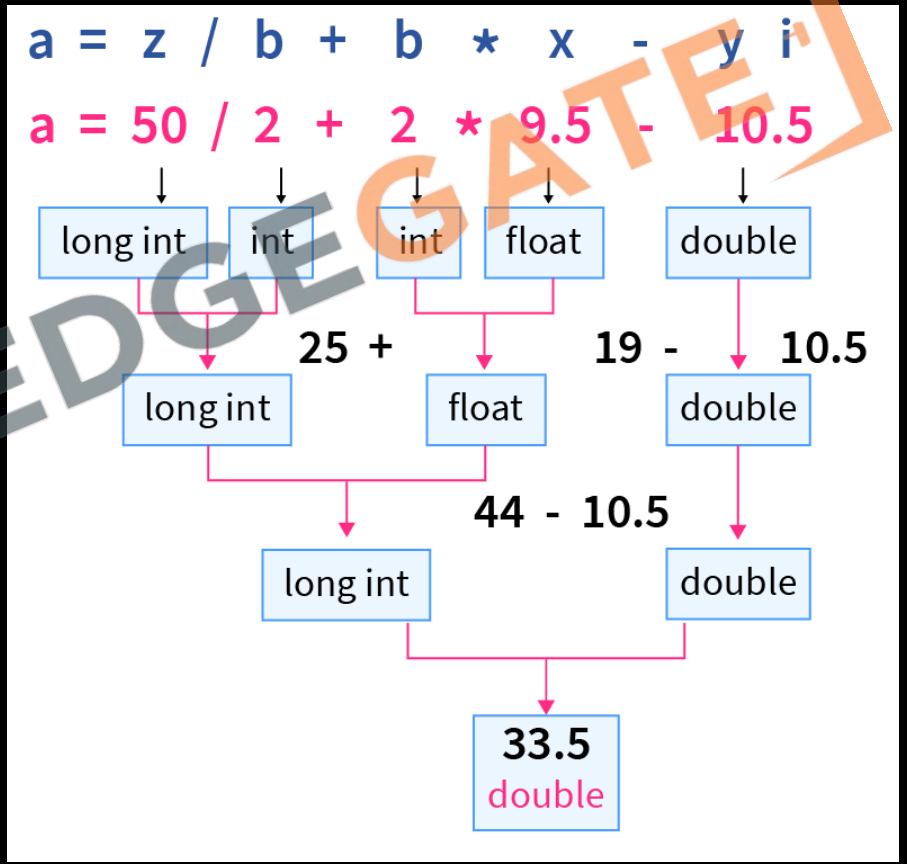
3.6 Type Conversions

Arithmetic Instruction	Result	Arithmetic Instruction	Result
k = 2 / 9	0	a = 2 / 9	0.0
k = 2.0 / 9	0	a = 2.0 / 9	0.2222
k = 2 / 9.0	0	a = 2 / 9.0	0.2222
k = 2.0 / 9.0	0	a = 2.0 / 9.0	0.2222
k = 9 / 2	4	a = 9 / 2	4.0
k = 9.0 / 2	4	a = 9.0 / 2	4.5
k = 9 / 2.0	4	a = 9 / 2.0	4.5
k = 9.0 / 2.0	4	a = 9.0 / 2.0	4.5

3.6 Type Conversions

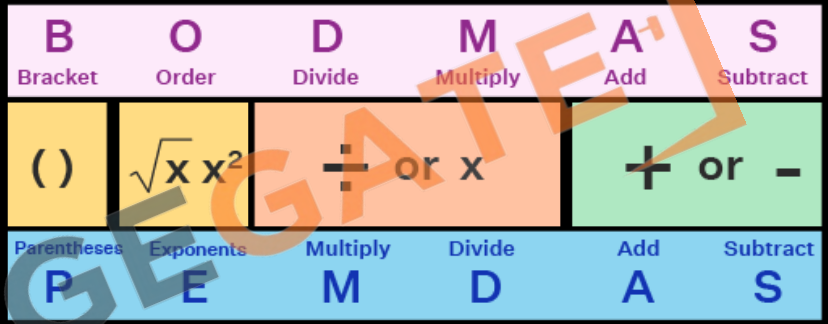
```c
int a,b = 2;
float x = 9.5;
double y = 10.5;
long int z = 50;
double d;
a = z/b+b*x-y;
printf("integer : %d\n",a);
return 0;
```

```
[Running] cd "/Users/prashantjain/
Desktop/test/"test
When stored as integer : 33
```



3.7 Hierarchy of Operations

Priority	Operators	Description
1st	* / %	multiplication, division, modular division
2nd	+ -	addition, subtraction
3rd	=	assignment



Example 1: Determine the hierarchy of operations and evaluate the following expression:

i = 2 * 3 / 4 + 4 / 4 + 8 - 2 + 5 / 8

Stepwise evaluation of this expression is shown below:

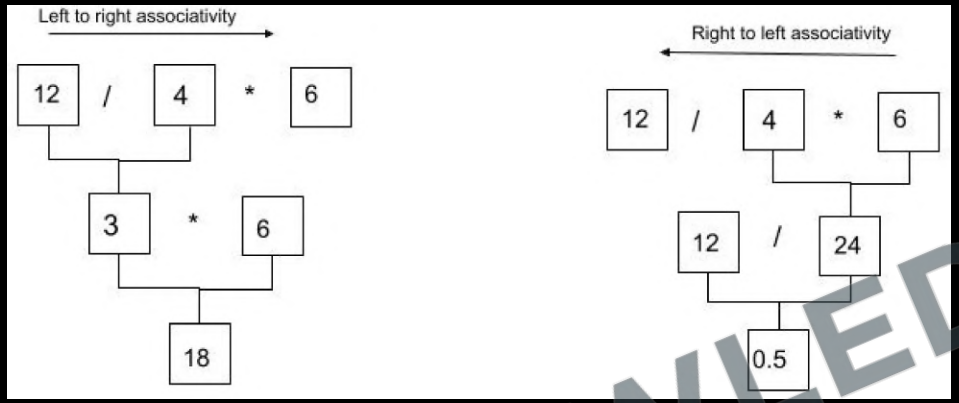
i = 2 * 3 / 4 + 4 / 4 + 8 - 2 + 5 / 8

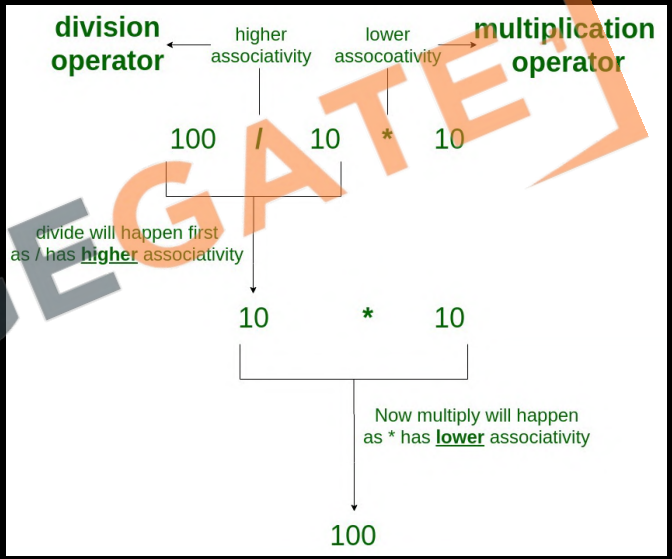
i = 6 / 4 + 4 / 4 + 8 - 2 + 5 / 8	operation: *
i = 1 + 4 / 4 + 8 - 2 + 5 / 8	operation: /
i = 1 + 1+ 8 - 2 + 5 / 8	operation: /
i = 1 + 1 + 8 - 2 + 0	operation: /
i = 2 + 8 - 2 + 0	operation: +
i = 10 - 2 + 0	operation: +
i = 8 + 0	operation: -
i = 8	operation: +

3.7 Hierarchy of Operations

Algebric Expression	C Expression
$a \times b - c \times d$	a * b – c * d
$(m + n)\ (a + b)$	(m + n) * (a + b)
$3x2 + 2x + 5$	3 * x * x + 2 * x + 5
$\frac{a+b+c}{d+e}$	(a + b + c) / (d + e)
$\left[\frac{2BY}{d+1}\ \frac{x}{3(z+y)}\right]$	2 * b * y / (d + 1) – x / 3 * (z + y)

3.8 Associativity of Operations





Operator Precedence: Determines the evaluation order of operators in an expression based on their priority levels.

Associativity: Defines the order of operation for operators with the same precedence, usually left-to-right or right-to-left.

3.8 Associativity of Operations

Operator Type	Category	Precedence	Associativity
Unary	postfix	a++, a--	Right to left
	prefix	++a, --a, +a, -a, ~, !	Right to left
Arithmetic	Multiplication	*, /, %	Left to Right
	Addition	+, -	Left to Right
Shift	Shift	<<, >>, >>>	Left to Right
Relational	Comparison	<, >, <=, >=, instanceOf	Left to Right
	equality	==, !=	Left to Right
Bitwise	Bitwise AND	&	Left to Right
	Bitwise exclusive OR	^	Left to Right
	Bitwise inclusive OR	|	Left to Right
Logical	Logical AND	&&	Left to Right
	Logical OR	||	Left to Right
Ternary	Ternary	? :	Right to Left
Assignment	assignment	=, +=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=, >>>=	Right to Left

3.9 Shorthand Operators

Operator symbol	Name of the operator	Example	Equivalent construct
+=	Addition assignment	x += 4;	x = x + 4;
-=	Subtraction assignment	x -= 4;	x = x - 4;
*=	Multiplication assignment	x *= 4;	x = x * 4;
/=	Division assignment	x /= 4;	x = x / 4;
%=	Remainder assignment	x %= 4;	x = x % 4;

3.10 Unary Operators

Operator	Description	Example
-	Converts a positive value to a negative	x = -y
Pre Increment	Increment the value by 1 and then use it in our statement	x = ++y
Pre Decrement	Decrement the value by 1 and then use it in our statement	x = --y
Post Increment	Use current value in the statement and then increment it by 1	x = y++
Post Decrement	Use current value in the statement and then decrement it by 1	x = y--

3.11 Control Instructions

Control Instructions enable us to specify the order in which various instructions in a program are executed.

1. Sequence Control: Executes instructions in the order they're written.
2. Selection Control: Chooses which instructions to run based on a condition (e.g., `if-else`).
3. Loop Control: Repeats instructions until a condition changes (e.g., `for`, `while`).
4. Case Control: Executes instructions from multiple options based on a variable's value (e.g., `switch`).

Revision

1. C Instructions
2. Type Declaration Instruction
3. Arithmetic Operators
4. Arithmetic Instruction
5. Integer and Float Conversions
6. Type Conversions
7. Hierarchy of Operations
8. Associativity of Operations
9. Shorthand Operators
10. Unary Operators
11. Control Instructions

CHALLENGE

11. Create a program that takes two numbers and shows result of all arithmetic operators (+,-,*,/,%).
12. Given an integer value, convert it to a floating-point value and print both.
13. Create a program to calculate product of two floating points numbers.
14. Create a program to calculate Perimeter of a rectangle.
    Perimeter of rectangle ABCD = A+B+C+D
15. Create a program to calculate the Area of a Triangle.
    Area of triangle = ½*B*H
16. Create a program to calculate simple interest.
    Simple Interest = (P x T x R)/100
17. Create a program to calculate Compound interest.
    Compound Interest = P(1 + R/100)t
18. Create a program to convert Fahrenheit to Celsius
    °C = (°F - 32) × 5/9

KG Coding

Some Other One shot Video Links:

- Complete Java
- Complete HTML & CSS
- Complete JavaScript
- Complete React and Redux
- One shot University Exam Series

http://www.kgcoding.in/

Our YouTube Channels

KG Coding Android App

KG Coding

Knowledge GATE

KG Placement Prep

Sanchit Socket

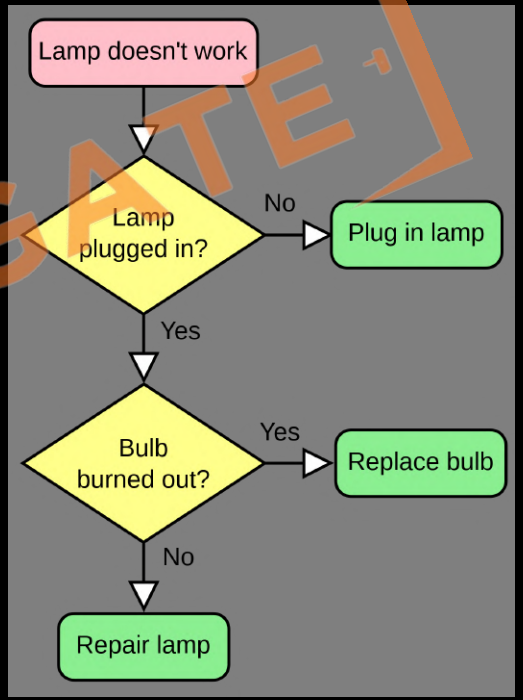
4. Decision Control Structure

if-else, switch, goto, ternary

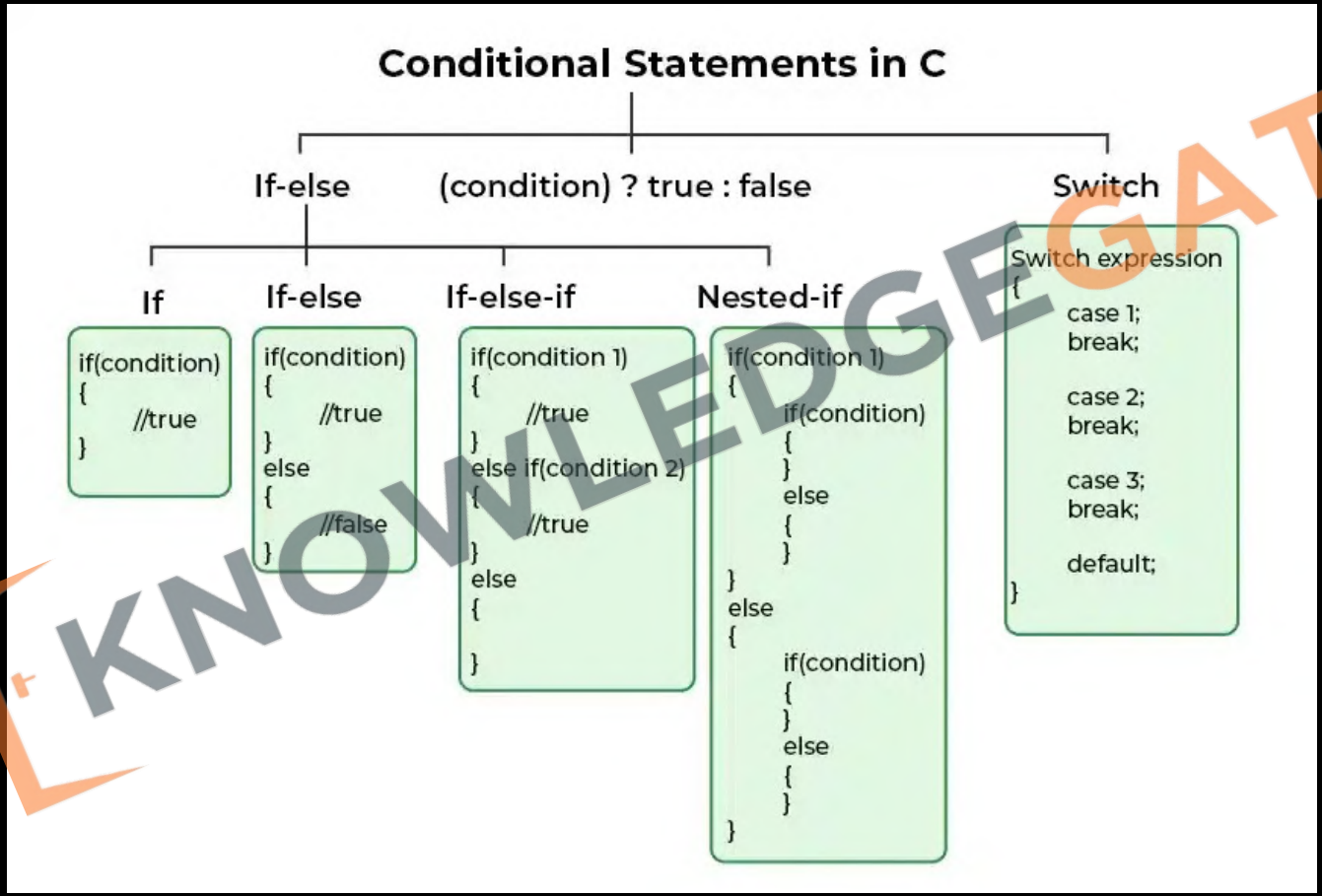
1. What is Decision Control ?
2. Relational Operators
3. if Statement
4. Truthy vs Falsy
5. if-else
6. if-else-if Ladder
7. Nested if
8. Logical Operators
9. Ternary Operator
10. Switch
11. Goto Statement

4.1 What is Decision Control ?

1. Conditional Execution: They allow code to run based on specific conditions, making programs dynamic.
2. Handles Complexity: Enables handling complex decisions through nested statements.
3. Enhances Flexibility: Increases the adaptability of programs to different scenarios.



4.1 What is Decision Control ?



4.2 Relational Operators

< , > , <= , >= , == , !=

Equality

- == Checks value equality.

Inequality

- != Checks value inequality.

Relational

- > Greater than.
- < Less than.
- >= Greater than or equal to.
- <= Less than or equal to.

Order of Relational operators is less than arithmetic operators

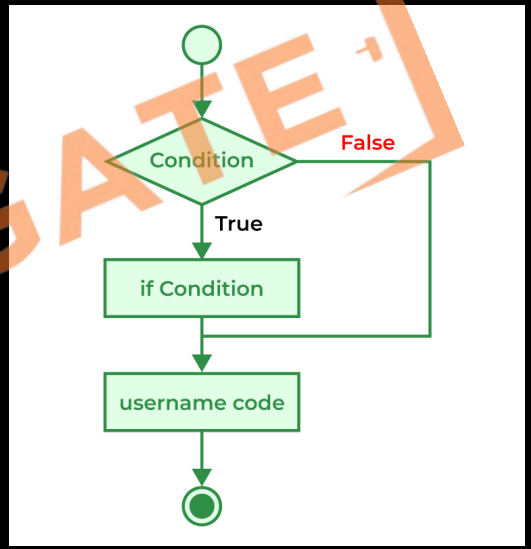
4.3 if Statement

Expression is true.

```
int test = 5;

if (test < 10)
{
  // codes
}

// codes after if
```

Expression is false.

```
int test = 5;

if (test > 10)
{
  // codes
}

// codes after if
```



1. Syntax: Uses if () {} to check a condition.
2. What is if: Executes block if condition is true, skips if false.
3. Curly Braces can be omitted for single statements, but not recommended.
4. Use Variables: Can store conditions in variables for use in if statements.

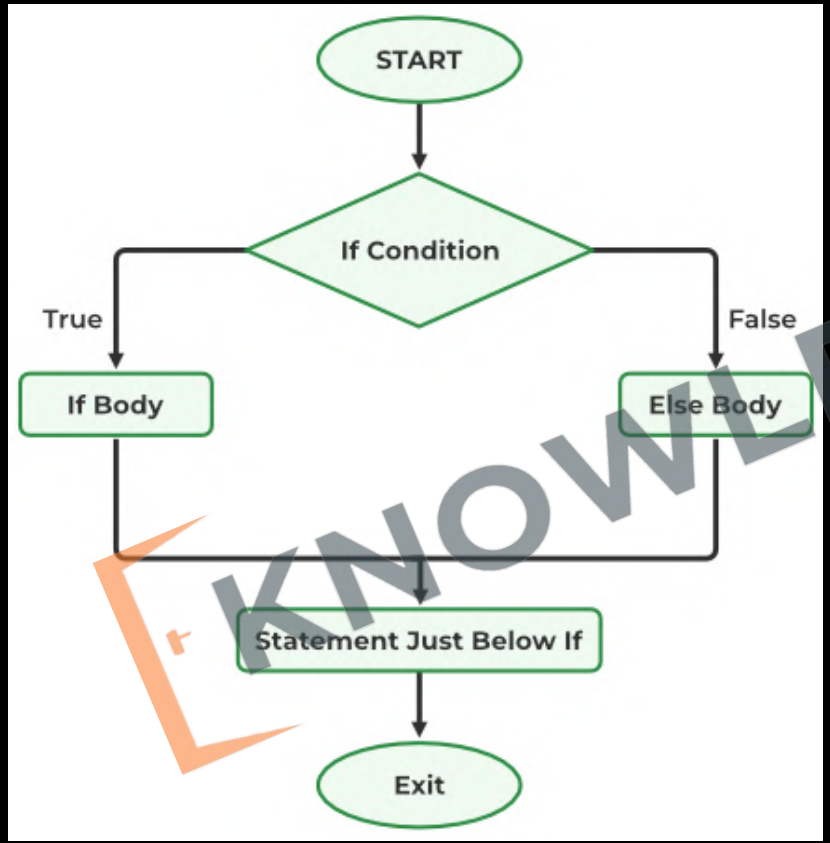
4.4 Truthy vs Falsy

```
if (1) {
    console.log("the value is truthy");
}
else {
    console.log("the value is falsy");
}
```

```
if (x) {
    // x is "truthy"
} else {
    // x is "falsy"
}
```


1. Falsy: 0 and NULL are considered false.
2. Truthy: Any non-zero value is true.
3. Use in Conditions: Utilized in if, while, and for to guide program flow.
4. Implicit Conversion: Non-boolean values are automatically considered true or false based on their numeric value.

4.5 if-else




```
if thirsty {
```

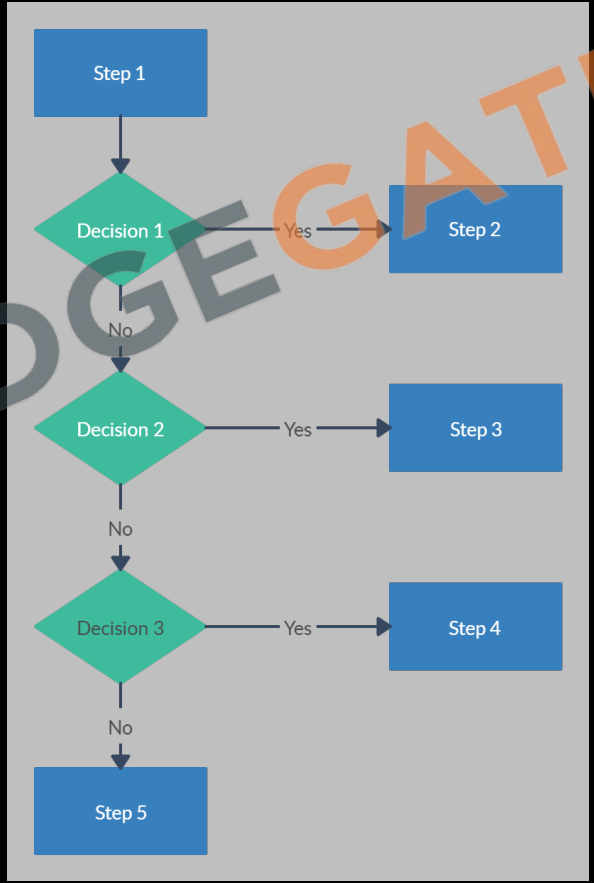
```
} else {
```

```
}
```

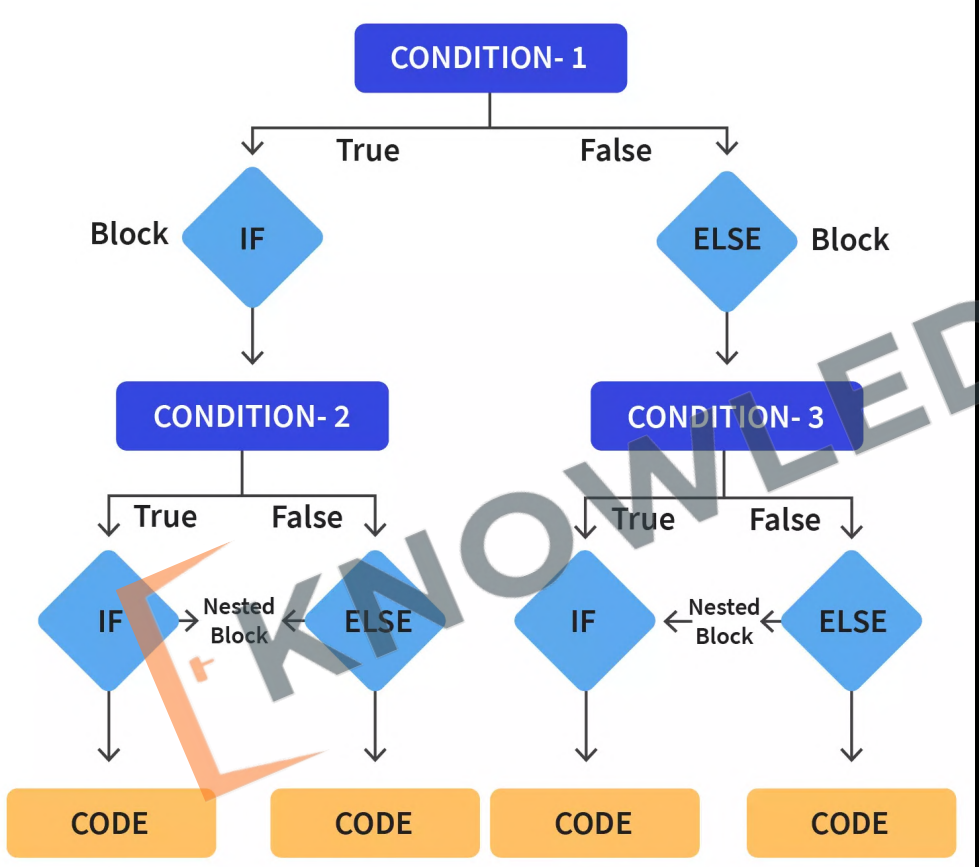
What is else: Executes a block when the if condition is false.

4.6 if-else-if Ladder

1. Sequential Checking: The if-else ladder checks multiple conditions one after the other, from top to bottom.
2. First True Condition: Executes the block of code associated with the first true condition it encounters.
3. Exits After Execution: Once a true condition's code block is executed, it exits the ladder and skips the remaining conditions.
4. Fallback with Else: If none of the conditions are true, the final else block (if present) executes as a default case.



4.7 Nested if



1. Condition Hierarchy: Enables hierarchical condition checks.
2. Complexity: Allows for detailed decision-making paths.
3. Syntax: An if inside another if or else.
4. Readability: Deep nesting can reduce code clarity.

4.8 Logical Operators

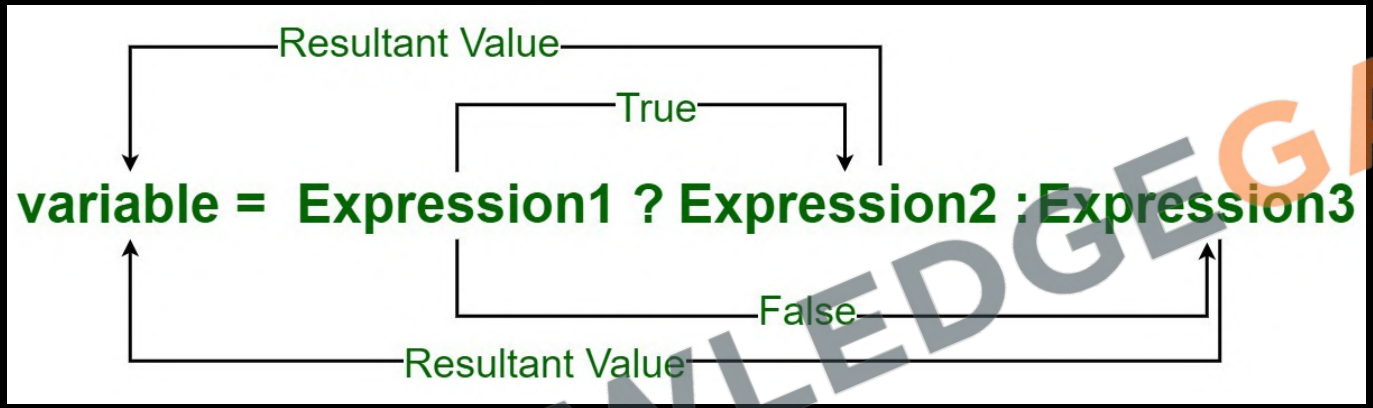

AND


Or

NOT

1. Types: && (AND), || (OR), ! (NOT)
2. AND (&&): All conditions must be true for the result to be true.
3. OR (||): Only one condition must be true for the result to be true.
4. NOT (!): Inverts the value of a condition.
5. Lower Priority than Arithmetic and Comparison operators

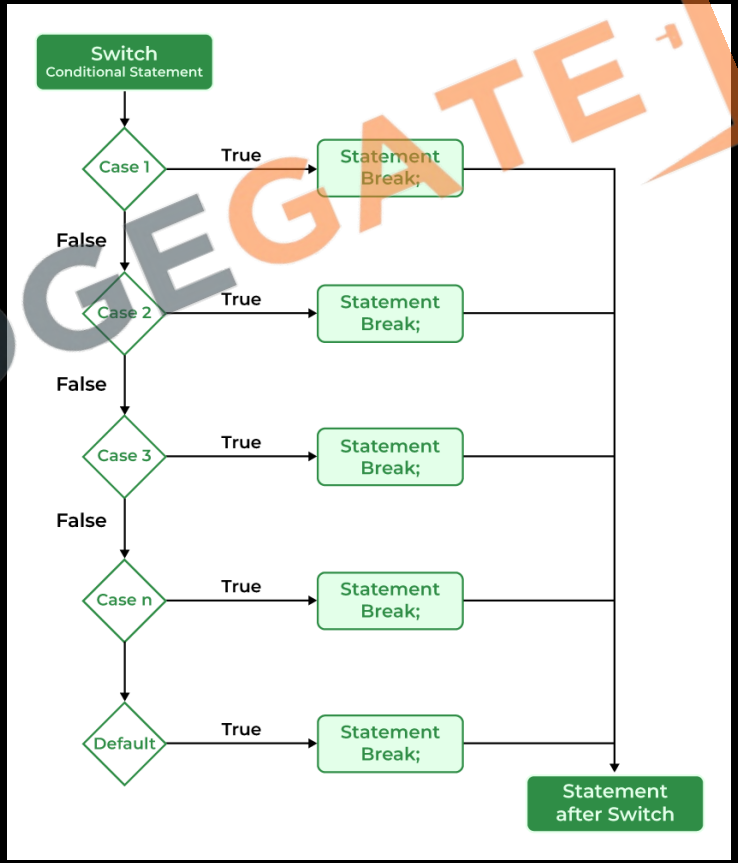
4.9 Ternary Operator



1. Syntax: condition ? expression1 : expression2
2. Condition: Boolean expression, evaluates to true or false.
3. Expressions: Both expressions must return compatible types.
4. Use Case: Suitable for simple conditional assignments.
5. Readability: Good for simple conditions, but can reduce clarity if overused.

4.10 Switch

1. Multi-way Branching: switch provides a cleaner method for multi-way branching than multiple if-else statements when testing the same expression.
2. Integer or Character Expressions: The switch expression must result in an integer or character value, used to match case labels.
3. Case Labels: Represents individual branches. Execution jumps to the matching case label.
4. Break Statement: Typically used to exit the switch block after a case is executed to prevent "fall through" to subsequent cases.
5. Default Case: Optional. Executes if no case matches. Placed at the end of the switch block.
6. Enhances Readability: For certain types of conditional logic, switch can make the code more readable compared to nested if-else statements.



4.10 Switch

```c
int main() {
    int dayNumber;
    printf("Enter a day number (1 to 7): ");
    scanf("%d", &dayNumber);

    switch(dayNumber) {
        case 1:
            printf("Sunday\n");
            break;
        case 2:
            printf("Monday\n");
            break;
        case 3:
            printf("Tuesday\n");
            break;
        case 4:
            printf("Wednesday\n");
            break;
        case 5:
            printf("Thursday\n");
            break;
        case 6:
            printf("Friday\n");
            break;
        case 7:
            printf("Saturday\n");
            break;
        default:
            // If the input is not between 1 and 7
            printf("Invalid day number.\n");
    }
    return 0;
}
```

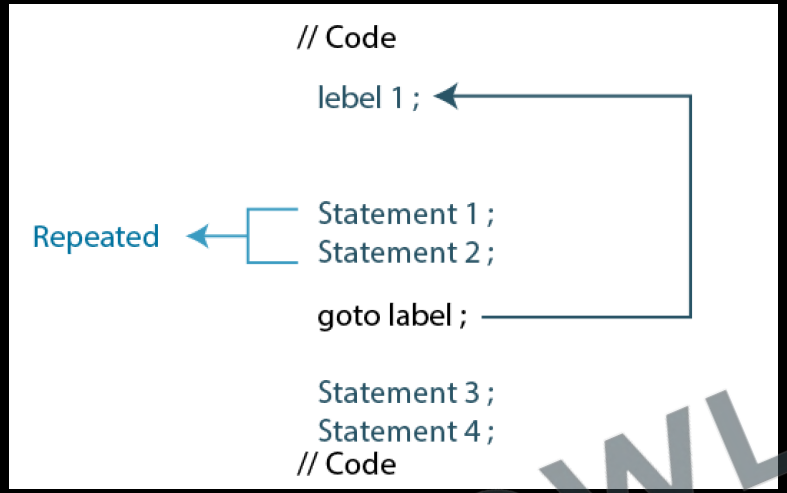
4.10 Switch vs If-else

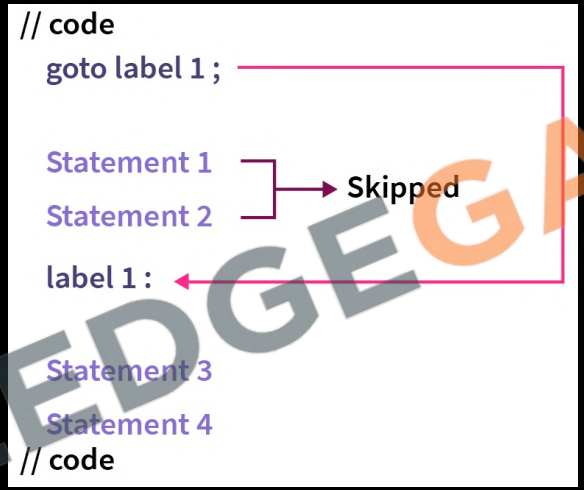
1. Scope of Comparison: switch tests equality for one variable; if-else can evaluate complex conditions.
2. Float expression cannot be tested using a switch.
3. Multiple cases cannot use same expressions.
4. Readability: switch is cleaner for many discrete values; if-else is more versatile.
5. Performance: switch might be faster for many cases due to compiler optimizations.
6. Fall-through: switch allows case fall-through; if-else executes only one matched block.

```
switch (i)
{
   case 1: statement_1;
           break;
   case 2: statement_2;
   case 3: statement_3;
           break;
   case 4: statement_4;
   case 5: statement_5;
   case 6: statement_6;
}
```

```
if (i == 1)
    statement_1;
else if (i == 2) {
    statement_2;
    statement_3;
}
else if (i == 3)
    statement_3;

else if (i == 4) {
    statement_4;
    statement_5
    statement_6
}
else if (i == 5) {
    statement_5;
    statement_6;
}
else if (i == 6)
    statement_6;
```

4.11 Goto Statement





1. Unconditional Jump: goto directs the flow to a specific label.
2. Label Usage: Defined by a name and colon (e.g., myLabel:).
3. Limited Scope: Target label must be in the same function.
4. Selective Use: Useful for exiting nested loops or error handling, but generally discouraged.
5. Drawbacks: Can lead to hard-to-follow code.

Revision

1. What is Decision Control ?
2. Relational Operators
3. if Statement
4. Truthy vs Falsy
5. if-else
6. if-else-if Ladder
7. Nested if
8. Logical Operators
9. Ternary Operator
10. Switch
11. Goto Statement

CHALLENGE

19. Create a program that determines if a number is positive, negative, or zero.
20. Create a program that determines if a number is odd or even.
21. Create a program that determines the greatest of the three numbers.
22. Create a program that determines if a given year is a leap year (considering conditions like divisible by 4 but not 100, unless also divisible by 400).
23. Create a program that calculates grades based on marks
    A -> above 90% B -> above 75%
    C -> above 60% D -> above 30%
    F -> below 30%
24. Create a program that categorize a person into different age groups
    Child -> below 13 Teen -> below 20
    Adult -> below 60 Senior-> above 60

CHALLENGE

25. Create a program to find the minimum of two numbers using ternary operator.
26. Create a program to find if the given number is even or odd using ternary operator.
27. Create a program to calculate the absolute value of a given integer using ternary operator.
28. Create a program to Based on a student's score, categorize as "High", "Moderate", or "Low" using the ternary operator (e.g., High for scores > 80, Moderate for 50-80, Low for < 50).
29. Create a program to print the month of the year based on a number (1-12) input by the user.
30. Create a program to create a simple calculator that uses a switch statement to perform basic arithmetic operations like addition, subtraction, multiplication, and division.

KG Coding

Some Other One shot Video Links:

- Complete Java
- Complete HTML & CSS
- Complete JavaScript
- Complete React and Redux
- One shot University Exam Series

http://www.kgcoding.in/

Our YouTube Channels

KG Coding Android App

KG Coding

Knowledge GATE

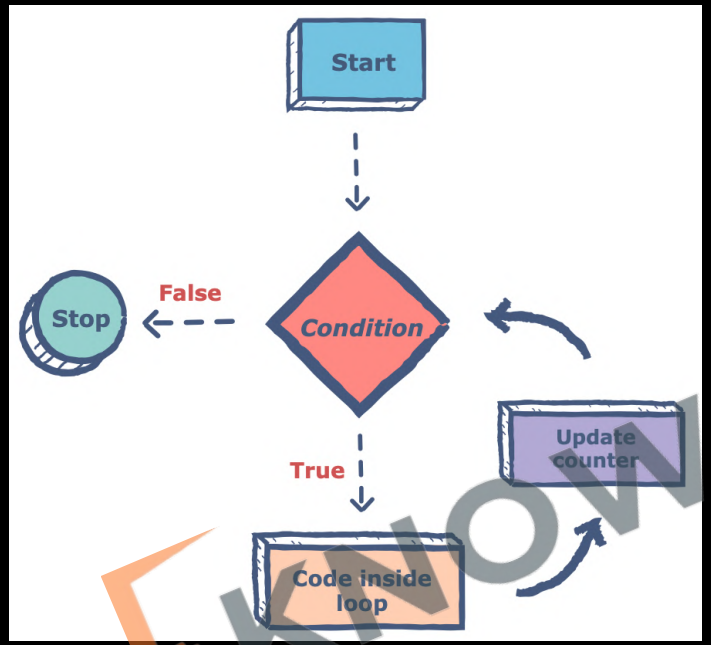
KG Placement Prep

Sanchit Socket

5 Iteration & Loop Control Structure

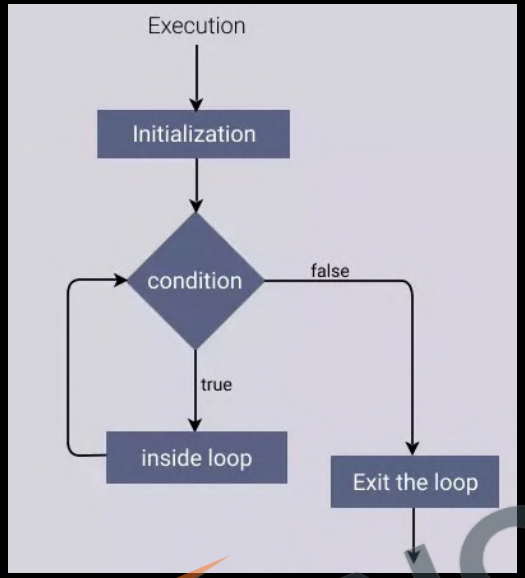
1. Need of loops
2. While Loop
3. For Loop
4. Break statement
5. Continue statement
6. Odd Loop
7. Do-while Loop
8. Infinite Loop

5.1 Need of loops



1. Code that runs multiple times based on a condition.
2. Repeated execution of code.
3. Loops automate repetitive tasks.
4. Types of Loops: while, for, while, do-while.
5. Iterations: Number of times the loop runs.

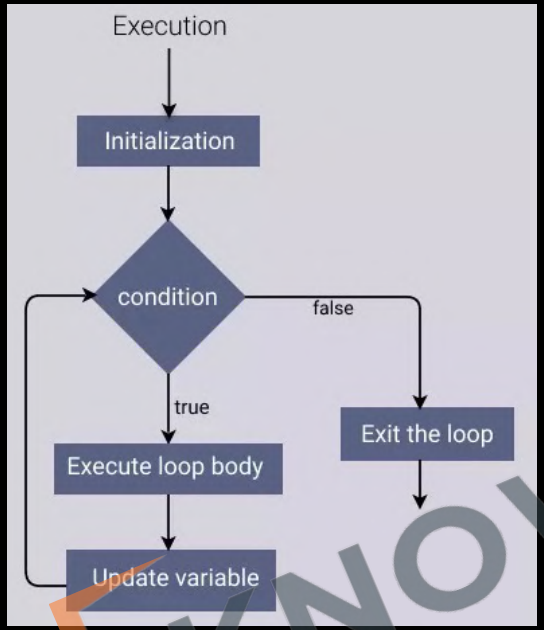
5.2 While Loop




```
while (condition) {
  // Body of the loop
}
```


1. Iterations: Number of times the loop runs.
2. Used for non-standard conditions.
3. Repeating a block of code while a condition is true.
4. Remember: Always include an update to avoid infinite loops.

5.3 For Loop

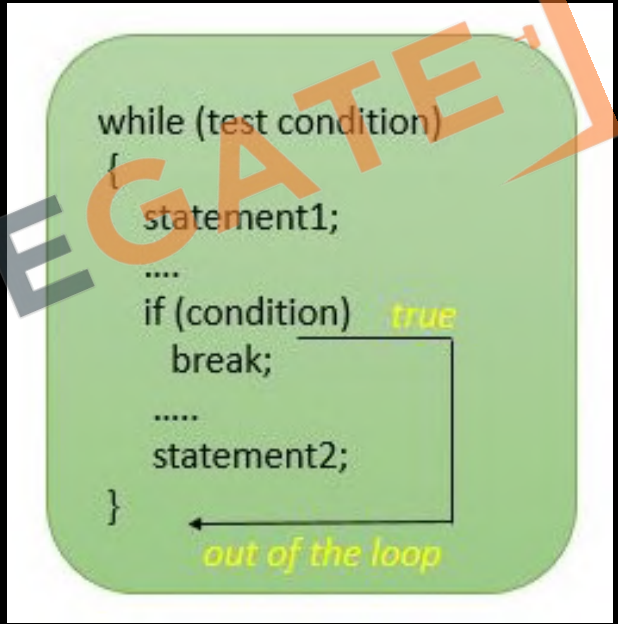



```
for (initialisation; condition; update) {
  // Body of the loop
}
```


1. Standard loop for running code multiple times.
2. Generally preferred for counting iterations.

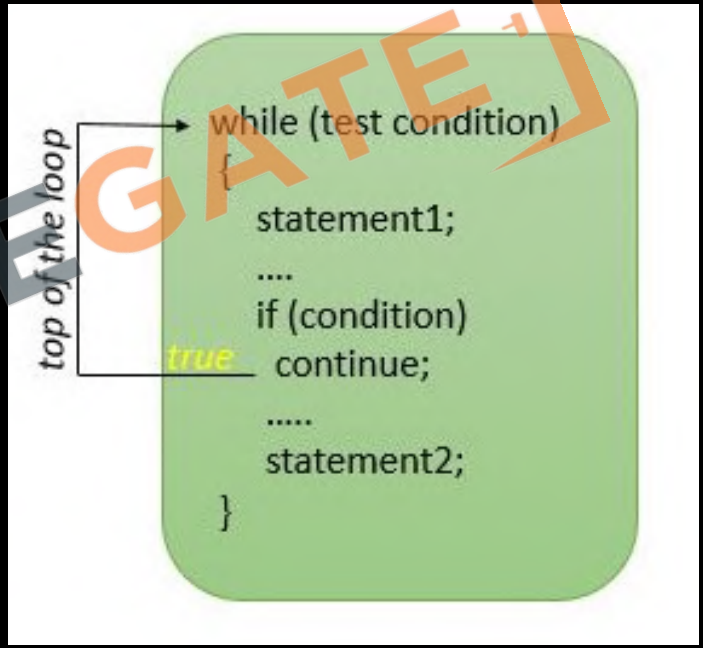
5.4 Break statement

1. Break lets you stop a loop early, or break out of a loop
2. Exits Loops: Ends for, while, do-while loops early.
3. Ends Switch Cases: Prevents fall-through in switch cases.
4. Immediate Effect: Immediately leaves the loop/switch.
5. Controls Flow: Alters program flow for efficiency.
6. Use Wisely: Important for readability.



5.5 Continue statement

1. Continue is used to skip one iteration or the current iteration
2. Next Iteration: Immediately starts the next iteration of the loop.
3. In while loop remember to do the increment manually before using continue.
4. Used in Loops: Works within for, while, do-while loops.
5. Not for switch: Unlike break, not used in switch statements.
6. Improves Logic: Helps in avoiding nested conditions within loops.

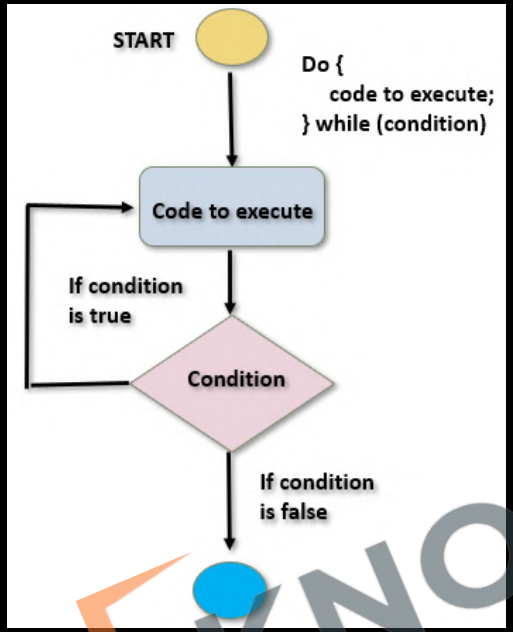


5.6 Odd Loop

```
/* Execution of a loop an unknown number of times */
main( )
{
      char  another ;
      int  num ;
      do
      {
            printf ( "Enter a number " ) ;
            scanf ( "%d", &num ) ;
            printf ( "square of %d is %d", num, num * num ) ;
            printf ( "\nWant to enter another number y/n " ) ;
            scanf ( " %c", &another ) ;
      } while ( another == 'y' ) ;
}
```

1. Condition-Driven: Run until a specific condition is fulfilled.
2. While and Do-While: Commonly used for indeterminate iterations.
3. Dynamic Iteration: Iterations depend on changing conditions or input.
4. Break Usage: May use break for exit in any loop type.
5. Practical Use: Ideal for processing with unknown completion point.
6. Design Carefully: Important to avoid infinite loops by ensuring a valid exit condition.

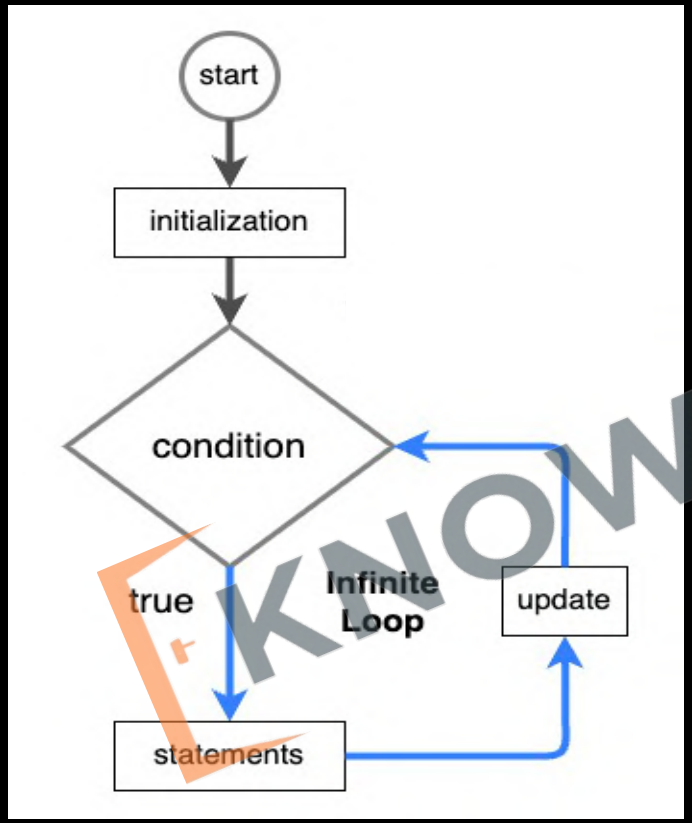
5.7 Do-while Loop




```c
do {
  // Body of the loop
}
while (condition);
```


1. Executes block first, then checks condition.
2. Guaranteed to run at least one iteration.
3. Unlike while, first iteration is unconditional.
4. Don't forget to update condition to avoid infinite loops.

5.8 Infinite Loop



1. Endless Execution: They run continuously.
2. Purposeful or Accidental: Used deliberately or by mistake.
3. Exit Strategy: Require break or similar statements for stopping.
4. Resource Intensive: May cause high CPU usage.

Revision

1. Need of loops
2. While Loop
3. For Loop
4. Break statement
5. Continue statement
6. Odd Loop
7. Do-while Loop
8. Infinite Loop

CHALLENGE

31. Develop a program that prints the multiplication table for a given number.
32. Create a program to sum all odd numbers from 1 to a specified number N.
33. Write a function that calculates the factorial of a given number.
34. Create a program that computes the sum of the digits of an integer.
35. Create a program to find the Least Common Multiple (LCM) of two numbers.
36. Create a program to find the Greatest Common Divisor (GCD) of two integers.
37. Create a program to check whether a given number is prime using while.
38. Create a program to reverse the digits of a number.
39. Create a program to print the Fibonacci series up to a certain number.
40. Create a program to check if a number is an Armstrong number.
41. Create a program to verify if a number is a palindrome.
42. Create a program that print patterns:

CHALLENGE

43. Create a program that prompts the user to enter a positive number. Use a do-while loop to keep asking for the number until the user enters a valid positive number.
44. Develop a program that calculates the sum of all numbers entered by a user until the user enters 0. The total sum should then be displayed.
45. Create a program using for loop multiplication table for a number.
46. Create a program using for to display if a number is prime or not.
47. Create a program using continue to sum all positive numbers entered by the user; skip any negative numbers.
48. Create a program using continue to print only even numbers using continue for odd numbers.
49. Write a program that continuously reads integers from the user and prints their squares. Use an infinite loop and a break statement to exit when a special number (e.g., -1) is entered.

KG Coding

Some Other One shot Video Links:

- Complete Java
- Complete HTML & CSS
- Complete JavaScript
- Complete React and Redux
- One shot University Exam Series

http://www.kgcoding.in/

Our YouTube Channels

KG Coding Android App

KG Coding

Knowledge GATE

KG Placement Prep

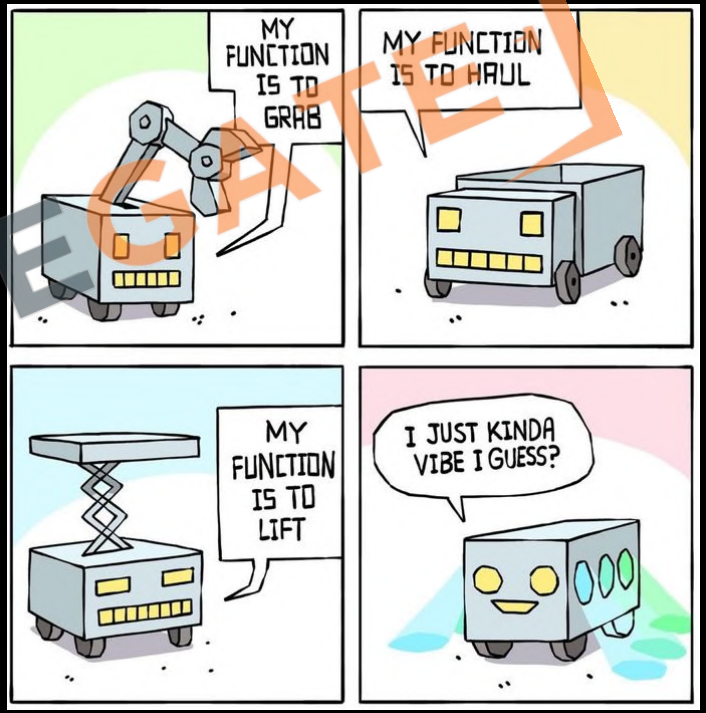
Sanchit Socket

6 Function and Recursion

1. What is a function
2. Function Prototype
3. Function Syntax / Definition
4. Function Call
5. Return Statement
6. Passing Values
7. Argument vs Parameter
8. Call by Value
9. Scoping Rule
10. Recursion

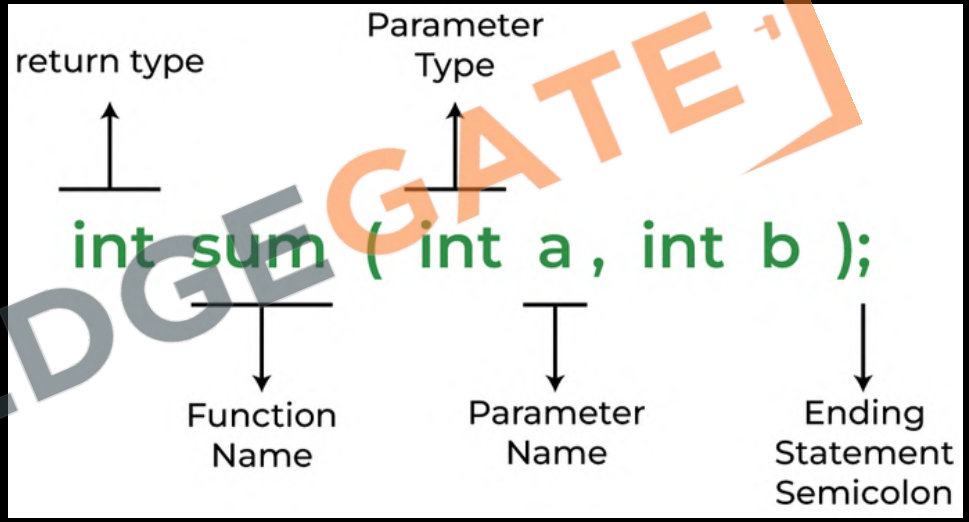
6.1 What is a function?

1. Definition: Blocks of reusable code.
2. DRY Principle: "Don't Repeat Yourself" it encourages code reusability.
3. Usage: Organizes code and performs specific tasks.
4. Naming Rules: Same as variable names: snake_case
5. Example: ”Beta Gas band kar de"



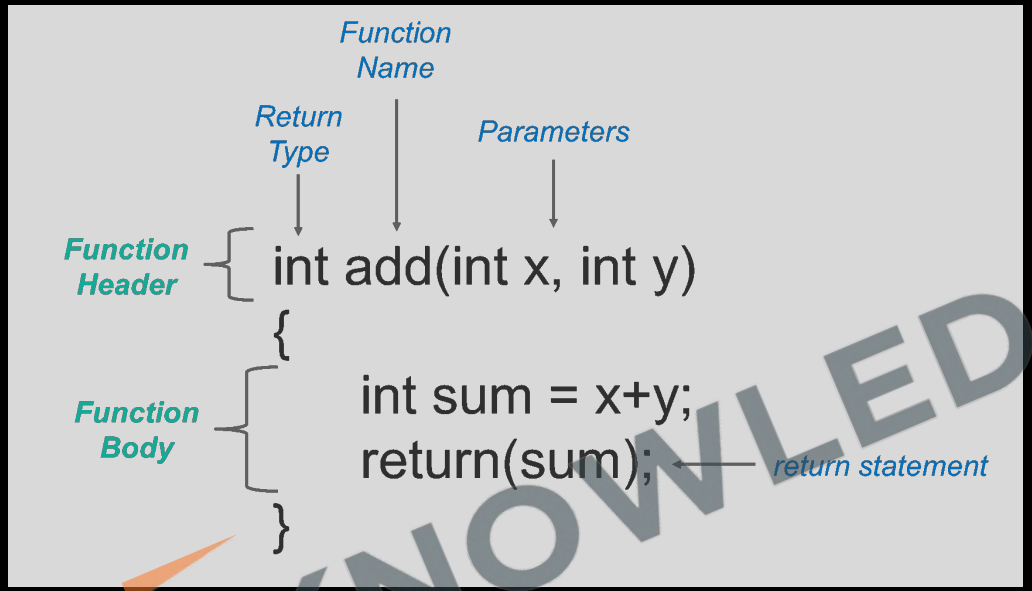
6.2 Function Prototype

```
void countTo100();  // Function Prototype
```



1. Definition: Specifies function name, return type, and parameters without the body.
2. Purpose: Enables type checking and forward declaration of functions.
3. Parameter Names Optional: Only types required in prototypes.
4. Placement: Often at the start of a C file or in a header file.

6.3 Function Syntax / Definition




```c
void countTo100() { // Function Definition
    for (int i = 1; i <= 100; i++) {
        printf("%d ", i);
    }
}
```

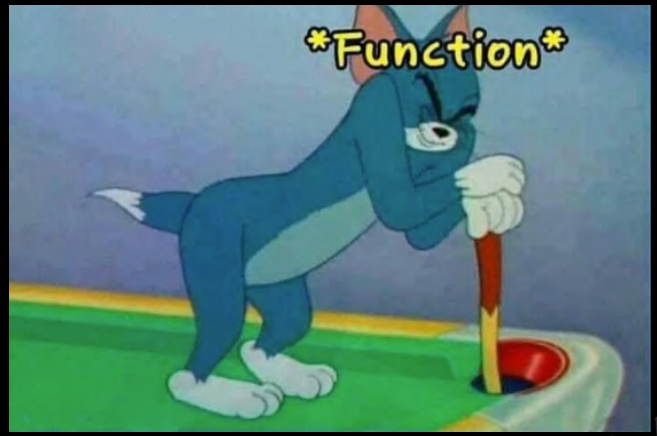

1. Follows same rules as variable names.
2. Use () to contain parameters.
3. Fundamental for code organization and reusability.

6.4 Function Call

```c
#include <stdio.h>

void countTo100();  // Function Prototype

int main() {
    countTo100();  // Function Call
    return 0;
}

void countTo100() { // Function Definition
    for (int i = 1; i <= 100; i++) {
        printf("%d ", i);
    }
}
```

Invoke by using the function name followed by ().

6.5 Return Statement



1. Sends a value back from a function.
2. Example: "Ek glass paani laao"
3. What Can Be Returned: Value, variable, calculation, etc.
4. Return ends the function immediately.
5. Function calls make code jump around.
6. Prefer returning values over using global variables.

6.6 Passing Values

Argument

Function

Return

1. Input values that a function takes.
2. Parameters put value into function, while return gets value out.
3. Example: "Ek packet dahi laao"
4. Naming Convention: Same as variable names.
5. Multiple Parameters: Functions can take more than one.

6.7 Argument vs Parameter

```c
#include <stdio.h>

int sum(int, int);  // Function Prototype

int main() {
    int first = 40, second = 50;
    int addition = sum(first, second);
    int add = sum(40, 50);
    return 0;
}

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

Arguments

Parameters

1. Parameters: Variables in a function definition, acting as placeholders.
2. Arguments: Actual values passed to a function at call time.

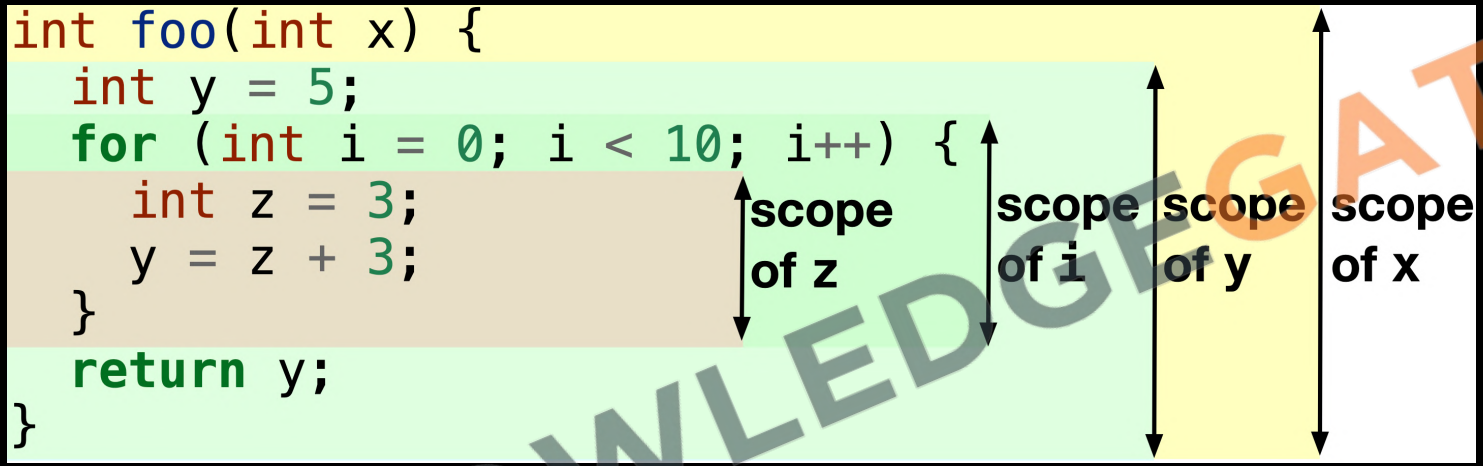
6.8 Call by Value

```c
#include <stdio.h>

// Function to try to swap two numbers
void trySwap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
    printf("Inside trySwap - a: %d, b: %d\n", a, b);
}

int main() {
    int x = 10, y = 20;

    printf("Before trySwap - x: %d, y: %d\n", x, y);
    trySwap(x, y); // Attempt to swap x and y
    // The original values are unchanged
    printf("After trySwap - x: %d, y: %d\n", x, y);
    return 0;
}
```

```
Before trySwap - x: 10, y: 20
Inside trySwap - a: 20, b: 10
After trySwap - x: 10, y: 20
```

1. Value Copy: Passes argument's copy, not the original.
2. Separate Memory: Parameters use distinct memory locations.
3. Data Safety: Original data remains unchanged by the function.
4. Direct Modification: Cannot modify original arguments directly.
5. Efficiency: Good for small data types, less so for large structures.
6. Ease of Use: Straightforward and safe for functions not altering inputs.

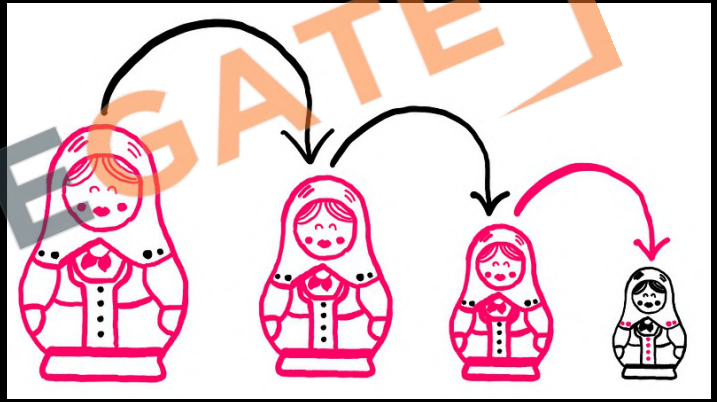
6.9 Scoping Rule



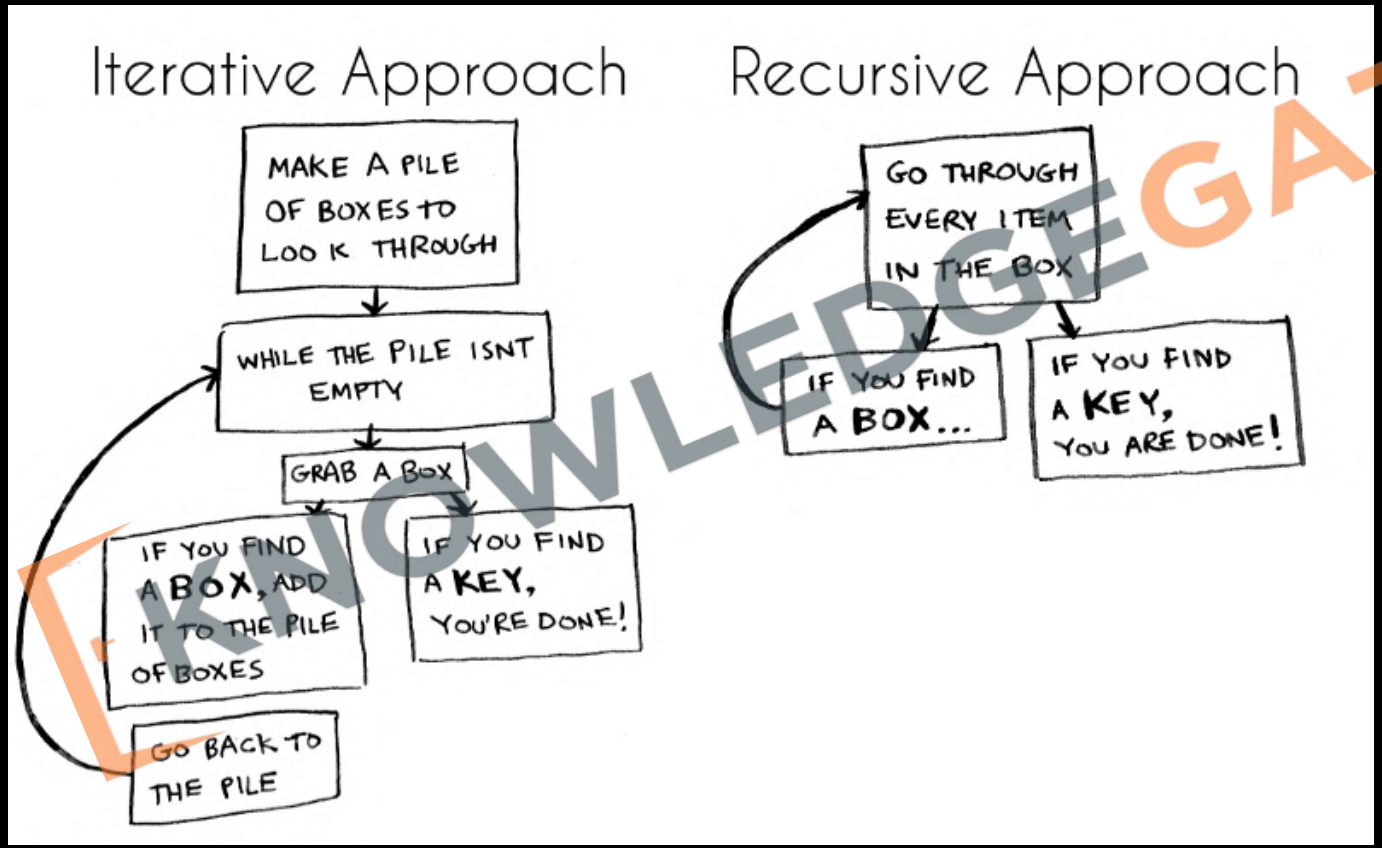
1. Local Scope: Variables inside functions are local, not accessible outside.
2. Global Scope: Variables outside functions are global, accessible anywhere in the file.
3. Block Scope: Variables in blocks (like if or loops) are accessible only within those blocks.
4. Function Parameters: Act as local variables within the function.
5. Lifetime: Local variables exist during function execution only.

6.10 Recursion

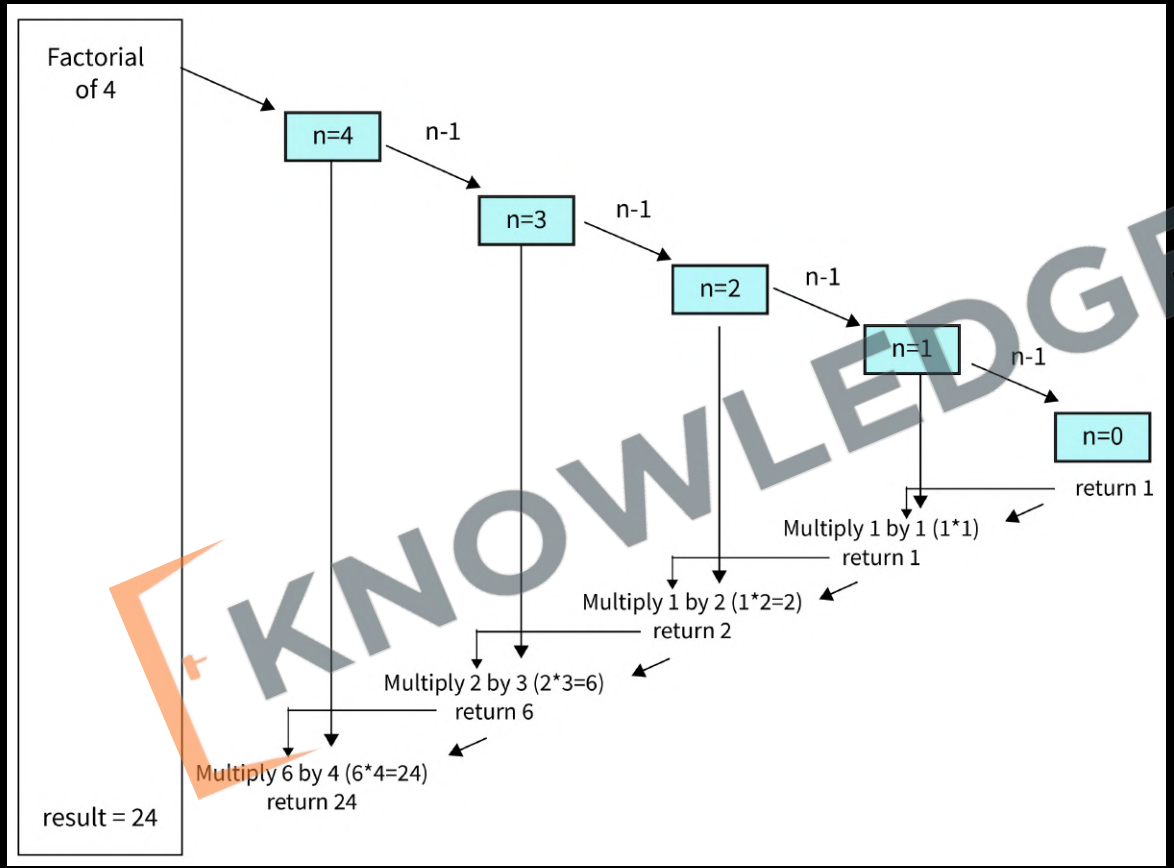
1. Self-Calling: A function that calls itself to solve sub-problems.
2. Base Case: Essential to stop recursion and prevent infinite loops.
3. Recursive Case: The condition under which the function keeps calling itself.
4. Stack Usage: Consumes stack space with each call, risk of overflow.
5. Applications: Ideal for divisible tasks, tree/graph traversals, sorting algorithms.
6. Memory Intensive: More overhead than iterative solutions.
7. Clarity: Often simplifies complex problems.



6.10 Iterative vs Recursive

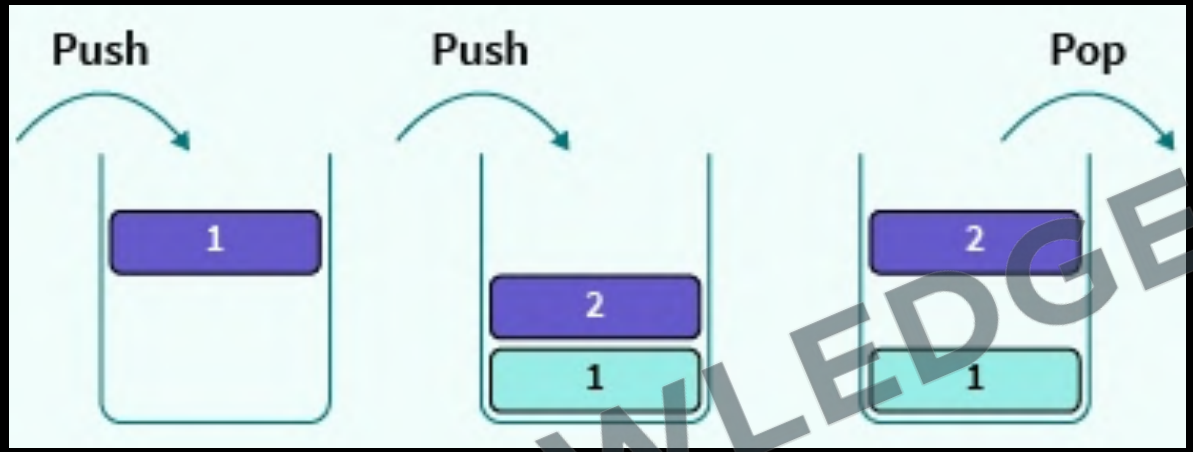


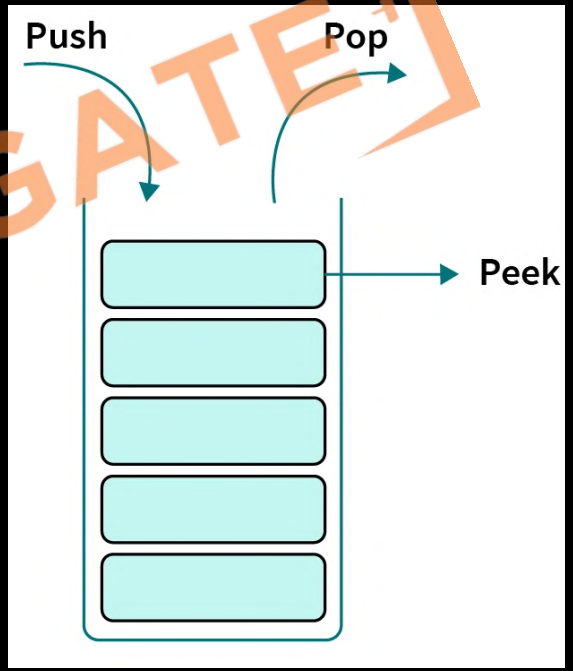
6.10 Factorial Using Recursion




```
// Recursive function to calculate factorial
long factorial(int n) {
    if (n == 0) // Base case
        return 1;
    else
        // Recursive case
        return n * factorial(n - 1);
}
```

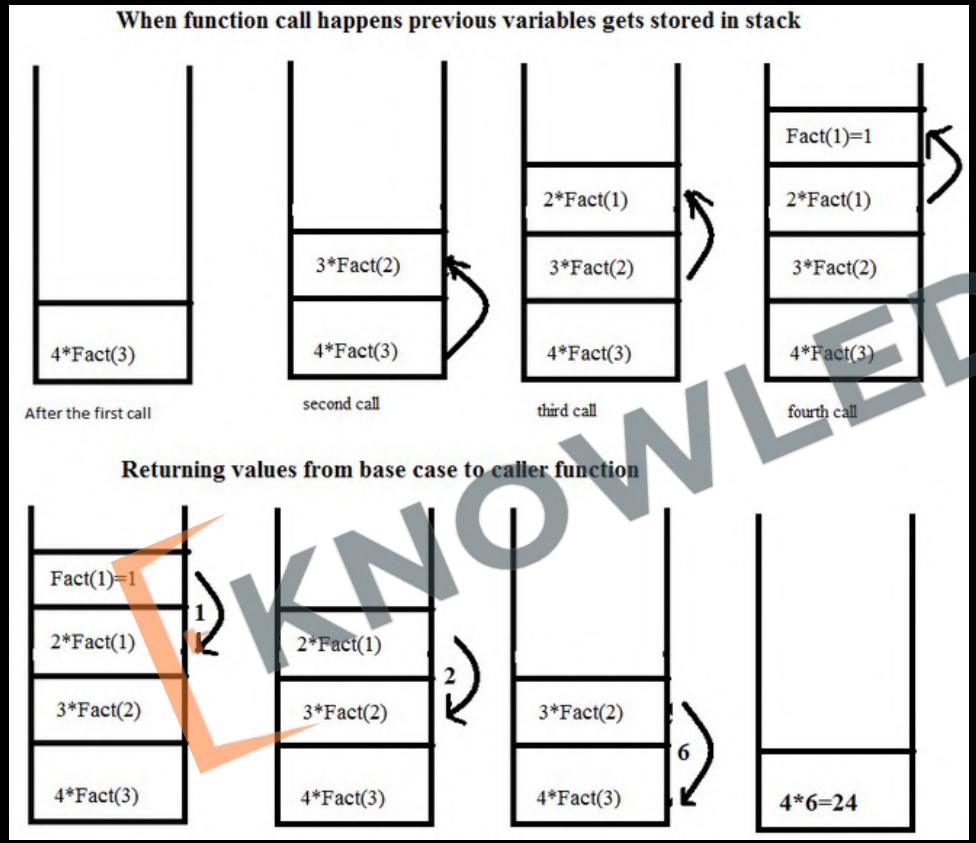
6.10 What is Stack





1. LIFO: Last-In, First-Out operation; the last element added is the first to be removed.
2. Operations: Mainly push (add an item) and pop (remove an item).
3. Top Element: Only the top element is accessible at any time, not the middle or bottom ones.
4. Overflow and Underflow: Overflow when full, underflow when empty during operations.

6.10 Recursion Stack



1. Memory Allocation: Recursive calls add frames to the call stack for variables and return points.
2. Growth with Depth: Deeper recursion equals more stack space.
3. Base Case: Essential to limit recursion depth and prevent stack overflow.
4. Stack Overflow Risk: Excessive recursion depth can crash the program.
5. Tail Recursion Optimization: Can reduce stack usage for certain recursive patterns.
6. Efficiency: Iterative solutions may be more stack-efficient than recursion for some problems.

Revision

1. What is a function
2. Function Prototype
3. Function Syntax / Definition
4. Function Call
5. Return Statement
6. Passing Values
7. Argument vs Parameter
8. Call by Value
9. Scoping Rule
10. Recursion

CHALLENGE

50. Write a function named greet that prints "Hello, World!" when called.
51. Write a function that adds that takes 4 int parameters and returns the sum.
52. Define a function square that takes an int and returns its square.
53. Call a function print_date that prints the current date. Define the function without any parameters.
54. Create a function max that takes two float arguments and returns the larger value.
55. Demonstrate with a function increment that the original integer passed to it does not change after incrementing it inside the function.
56. Call a function get_average that takes five int numbers and returns the average.
57. Create a program using recursion to display the Fibonacci series upto a certain number.
58. Create a program using recursion to check if a number is a palindrome using recursion.

KG Coding

Some Other One shot Video Links:

- Complete Java
- Complete HTML & CSS
- Complete JavaScript
- Complete React and Redux
- One shot University Exam Series

http://www.kgcoding.in/

Our YouTube Channels

KG Coding Android App

KG Coding

Knowledge GATE

KG Placement Prep

Sanchit Socket

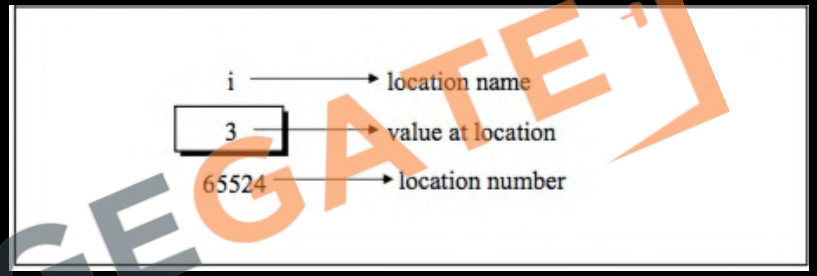
7 Pointers

1. Introduction to Pointers
2. & (Address) Operator
3. * (Value at address) Operator
4. Pointer Declaration
5. Pointer to a Pointer
6. Call By Reference

7.1 Introduction to Pointers

```c
int main() {
    int i = 3;
    printf("Value of i = %d", i);
    return 0;
}
```

```
[Running] cd "/Users/prashantjain/Desktop/t
Value of i = 3
[Done] exited with code=0 in 0.819 seconds
```

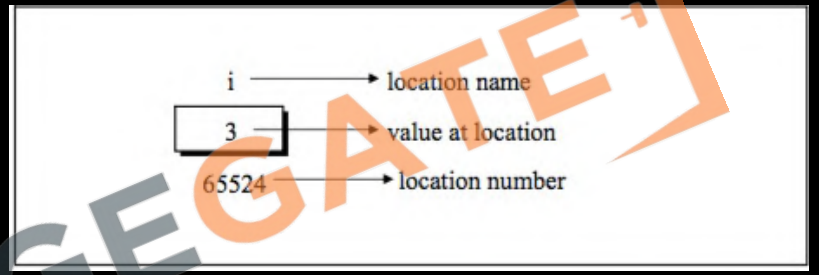


Address Storage: Pointers store memory addresses of other variables rather than data itself.

7.2 & (Address) Operator

```c
int main() {
    int i = 3;
    printf("Address of i = %p\n", &i);
    printf("Value of i = %d", i);
    return 0;
}
```

```
[Running] cd "/Users/prashantjain/Desktop/test/"
Address of i = 0x16f3db388
Value of i = 3
[Done] exited with code=0 in 0.397 seconds
```

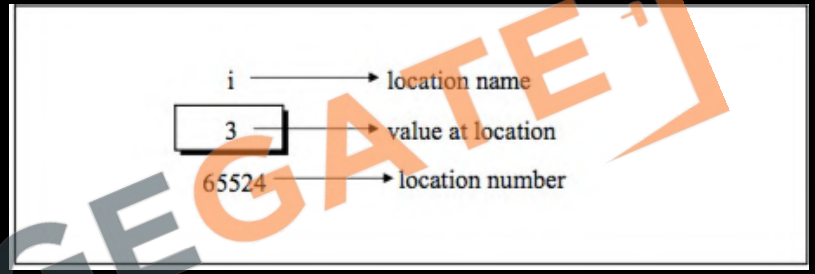


Address Operator: The ampersand & gets the address of a variable

7.3 * (Value at address) Operator

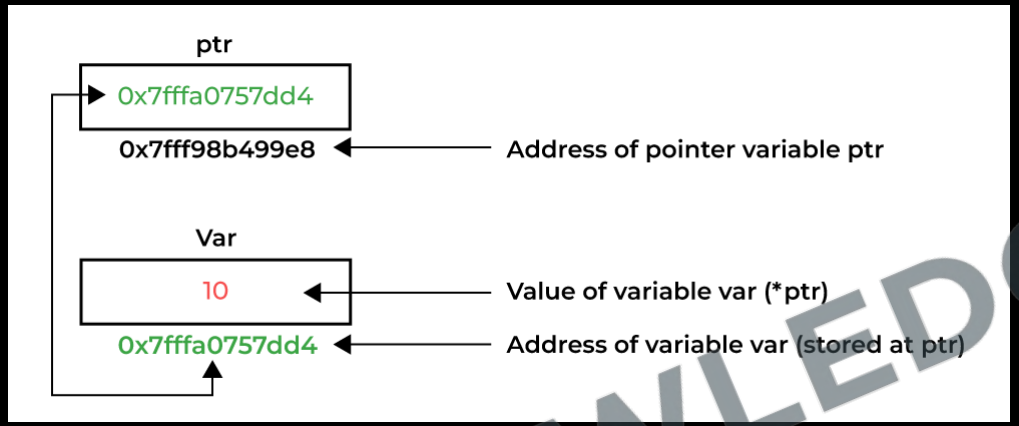
```c
int main() {
    int i = 3;
    printf("Address of i = %p\n", &i);
    printf("Value of i = %d\n", i);
    printf("Value of i = %d", *(&i));
    return 0;
}
```

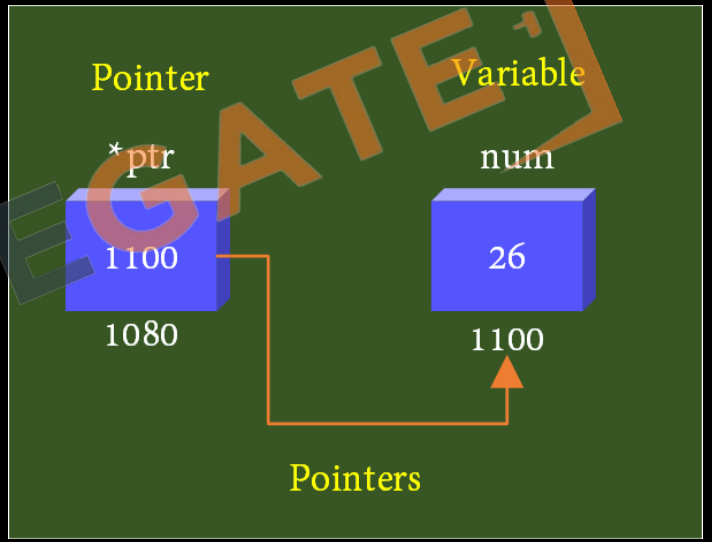
```
[Running] cd "/Users/prashantjain/Desktop/test/"
Address of i = 0x16f75b388
Value of i = 3
Value of i = 3
[Done] exited with code=0 in 0.395 seconds
```



* Operator is called Value at address operator. This operator is used to get the value from a particular address

7.4 Pointer Declaration





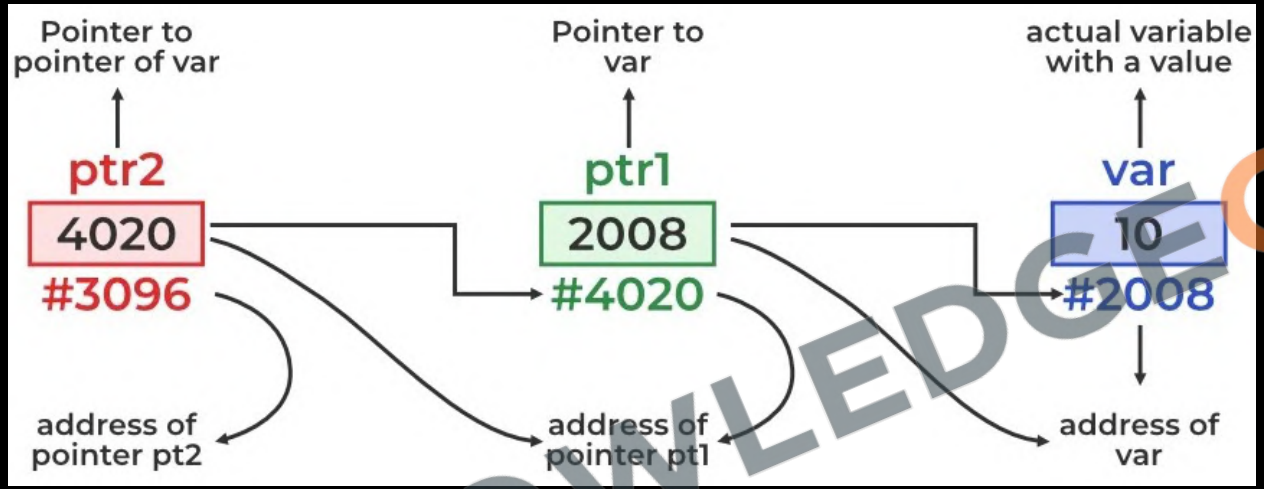
Declared with an asterisk * before the variable name, e.g., int *ptr;

7.4 Pointer Declaration

```c
int main() {
    int i = 3;
    int *j;

    j = &i;
    printf("Address of i = %p\n", &i);
    printf("Address of i = %p\n", j);
    printf("Address of j = %p\n", &j);
    printf("Value of j = %p\n", j);
    printf("Value of i = %d\n", i);
    printf("Value of i = %d\n", *(&i));
    printf("Value of i = %d", *j);
    return 0;
}
```

```
[Running] cd "/Users/prashantjain/Desktop/t
Address of i = 0x16da8b388
Address of i = 0x16da8b388
Address of j = 0x16da8b380
Value of j = 0x16da8b388
Value of i = 3
Value of i = 3
Value of i = 3
[Done] exited with code=0 in 0.361 seconds
```

7.5 Pointer to a Pointer



A pointer to a pointer is a type of pointer that stores the address of another pointer, allowing indirect access to the value of the variable the first pointer points to.

7.5 Pointer to a Pointer

```c
int main() {
    int i = 3, *j, **k;
    j = &i;
    k = &j;
    printf("Address of i = %p\n", &i);
    printf("Address of i = %p\n", j);
    printf("Address of i = %p\n", *k);
    printf("Address of j = %p\n", &j);
    printf("Address of j = %p\n", k);
    printf("Address of k = %p\n", &k);
    printf("Value of j = %p\n", j);
    printf("Value of k = %p\n", k);
    printf("Value of i = %d\n", i);
    printf("Value of i = %d\n", *(&i));
    printf("Value of i = %d\n", *j);
    printf("Value of i = %d", **k);
    return 0;
}
```

```
[Running] cd "/Users/prashan
Address of i = 0x16b017388
Address of i = 0x16b017388
Address of i = 0x16b017388
Address of j = 0x16b017380
Address of j = 0x16b017380
Address of k = 0x16b017378
Value of j = 0x16b017388
Value of k = 0x16b017380
Value of i = 3
Value of i = 3
Value of i = 3
Value of i = 3
[Done] exited with code=0 in
```

7.6 Call By Reference

```c
// Function to swap two numbers using pointers (call by reference)
void trySwap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
    printf("Inside trySwap - a: %d, b: %d\n", *a, *b);
}

int main() {
    int x = 10, y = 20;

    printf("Before trySwap - x: %d, y: %d\n", x, y);
    // Pass the addresses of x and y to the trySwap function
    trySwap(&x, &y); // Now the values of x and y will be swapped
    printf("After trySwap - x: %d, y: %d\n", x, y);

    return 0;
}
```

```
[Running] cd "/Users/prashantjain/Desktop/test/"
Before trySwap - x: 10, y: 20
Inside trySwap - a: 20, b: 10
After trySwap - x: 20, y: 10
```

1. Direct Access: Call by reference passes the address of variables, allowing functions to modify the actual values.
2. Pointers in C: Implemented using pointers that point to the original data.
3. Efficiency: Avoids copying large data structures, saving memory and time.
4. Multiple Returns: Can return multiple values from a function via out-parameters.
5. Risk: Increases the potential for unintended side effects if not used carefully.

Revision

1. Introduction to Pointers
2. & (Address) Operator
3. * (Value at address) Operator
4. Pointer Declaration
5. Pointer to a Pointer
6. Pointer Arithmetic
7. Call By Reference

CHALLENGE

59. Write a program that declares an integer variable and a pointer to it. Assign a value and print it using the pointer.
60. Write a program to change the value of an integer variable using a pointer and the * operator.
61. Declare a pointer to a char and use it to read and print a character entered by the user.
62. Implement a void minmax(int *a, int *b, int *min, int *max) function that takes two integer pointers a and b as input and assigns the smaller value to min and the larger value to max using call by reference. Write a main function to test it with different values.

KG Coding

Some Other One shot Video Links:

- Complete Java
- Complete HTML & CSS
- Complete JavaScript
- Complete React and Redux
- One shot University Exam Series

http://www.kgcoding.in/

Our YouTube Channels

KG Coding Android App

KG Coding

Knowledge GATE

KG Placement Prep

Sanchit Socket

8 Data Types and Storage Classes

1. Long
2. Signed and Unsigned
3. What are Storage Classes
4. Automatic
5. Register
6. Static
7. External
8. Usage of Storage Classes

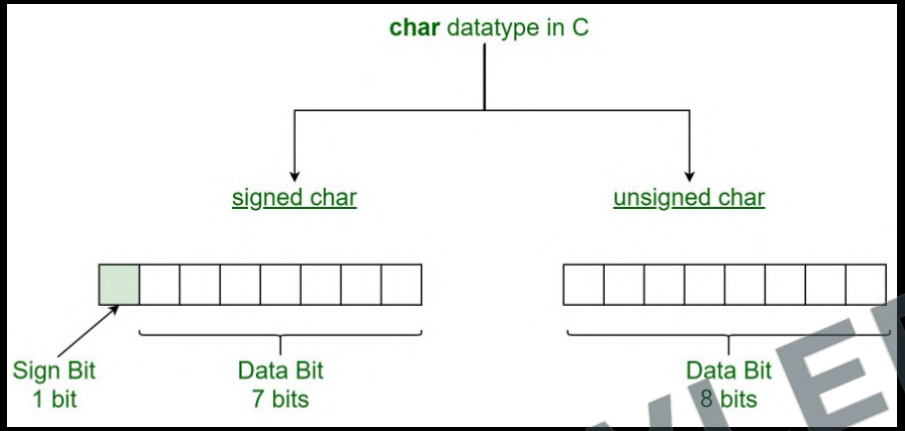
8.1 Long

```c
long factorial(int n) {
    if (n >= 1) {
        return n * factorial(n - 1);
    } else {
        return 1;
    }
}

int main() {
    // Factorial of 20 is beyond the range of int
    int number = 20;
    long result;
    result = factorial(number);
    printf("The factorial of %d is %ld\n",
     number, result);
    return 0;
}
```

```
[Running] cd "/Users/prashantjain/Desktop/test/"
The factorial of 20 is 2432902008176640000

[Done] exited with code=0 in 0.595 seconds
```

1. Size: Larger than int, often 64-bits on 64-bit systems and 32-bits on 32-bit systems.
2. Range: Can hold larger integer values, useful for extended arithmetic precision.
3. Suffix: Denoted by an L or l suffix for long literals, e.g., 100L.
4. Types: Has a long long variant for even larger integers, at least 64 bits.
5. Standard: Size guaranteed by the C standard to be at least 32 bits.

8.2 Signed and Unsigned



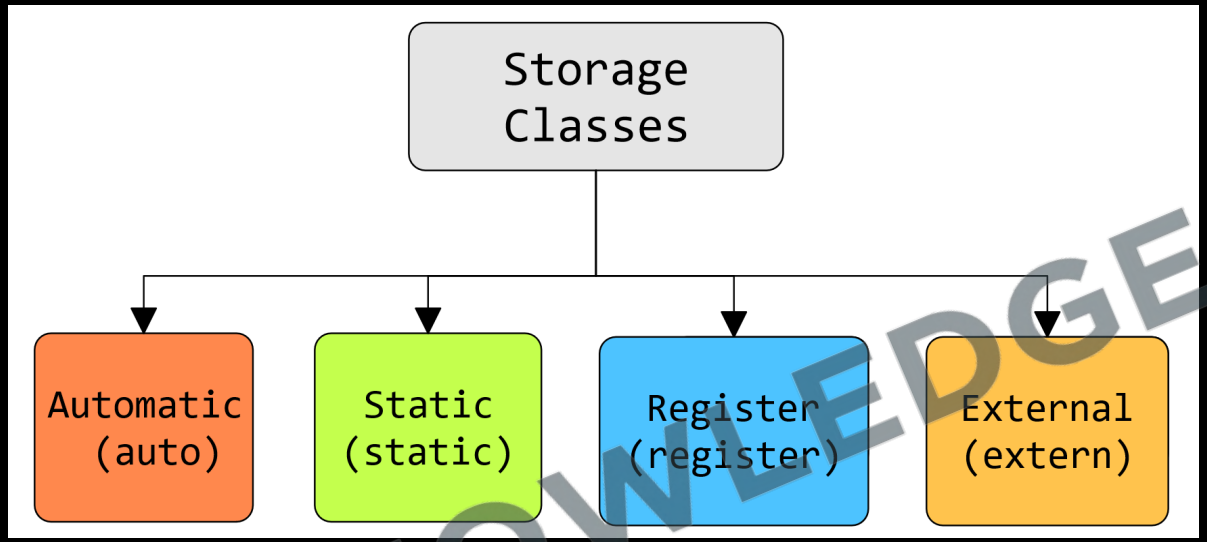
1. Signed int: Can represent both positive and negative numbers, including zero.
2. Unsigned int: Can only represent non-negative numbers, doubling the maximum positive value compared to signed.
3. Range: Signed int range is roughly from -2^31 to 2^31-1, while unsigned is from 0 to 2^32-1 on a 32-bit system.
4. Overflow: Unsigned ints wrap around on overflow.
5. Usage: Choose unsigned for countable quantities where negative values don't make sense.

8.2 Signed and Unsigned

```c
#include <stdio.h>

int main() {
    unsigned int length = 10;  // Length of the rectangle (can't be negative)
    unsigned int width = 5;    // Width of the rectangle (can't be negative)
    unsigned int area;         // Area of the rectangle

    int temperature = -15;     // Temperature (can be negative)

    // Calculate the area of the rectangle
    area = length * width;

    printf("The area of the rectangle is: %u square units\n", area);
    printf("The current temperature is: %d degrees Celsius\n", temperature);

    return 0;
}
```

```
[Running] cd "/Users/prashantjain/Desktop/test/"
The area of the rectangle is: 50 square units
The current temperature is: -15 degrees Celsius
```

8.3 What are Storage Classes



1. Lifetime Management: Storage classes determine the lifetime (scope and duration) of variables.
2. Scope Control: Dictate where a variable can be accessed, either within a function/block or throughout the program.
3. Memory Location: Influence where the variable is stored, such as in the stack, heap, or data segment.
4. Initialization: Define default initial values, like zero for static variables.

8.4 Automatic

```c
void function() {
    // An auto variable (though auto keyword is usually omitted)
    int autoVar = 10;
    printf("The value of autoVar is: %d\n", autoVar);
    // autoVar is destroyed here as it goes out of scope
}

int main() {
    function(); // Calls function with auto variable
    // function() has finished; autoVar is no longer accessible
    return 0;
}
```

```
[Running] cd "/Users/prashantjain/Desktop/test/"
The value of autoVar is: 10

[Done] exited with code=0 in 0.821 seconds
```

1. Default Class: Variables within a function are auto by default if no other storage class specifier is provided.
2. Stack Storage: auto variables are typically stored on the stack.
3. Function Scope: They are local to the block in which they are defined.
4. No Initial Value: Typically have an undefined initial value unless explicitly initialized.
5. Lifetime: Created when block is entered and destroyed when it is exited.

8.5 Register

```c
int main() {
    // Suggests storing counter in a register
    register int counter;
    for (counter = 0; counter < 5; ++counter) {
        printf("Counter is %d\n", counter);
    }

    // Note: Attempting to get the address of counter
    // like &counter, will cause a compile error
    return 0;
}
```

```
[Running] cd "/Users/prashantjain/Desktop/test/"
Counter is 0
Counter is 1
Counter is 2
Counter is 3
Counter is 4
```

1. CPU Register: Hints to the compiler that the variable should be stored in a CPU register for faster access.
2. Limited Size: Can only be applied to variables that can fit within a CPU register.
3. No Memory Address: Cannot use the address-of operator (&) on a register variable since it might not have a memory address.
4. Function Scope: Local to the function or block they are defined in, similar to auto.
5. Quick Access: Intended for variables that are heavily used, such as counters in loops.

8.6 Static

```c
void incrementCounter() {
    // A static variable retains its value between
    // function calls
    static int counter = 0;
    counter++;
    printf("Counter is %d\n", counter);
}

int main() {
    for (int i = 0; i < 3; i++) {
        // Each call will increment the counter
        incrementCounter();
    }
    // The counter retains its value between calls
    // and is not reinitialized
    return 0;
}
```

```
[Running] cd "/Users/prashantjain/Desktop/test/"
Counter is 1
Counter is 2
Counter is 3
```

1. Persistence: static variables retain their value between function calls.
2. Initialization: Automatically initialized to zero if no initial value is provided.
3. Function and File Scope: static variables can be local to a function or have file scope if declared outside of a function.
4. Internal Linkage: File-scope static variables are only visible within the file they are declared in, not externally.
5. Single Instance: The variable is allocated once and is not recreated with each function call.

8.7 External

```c
// main.c file
#include <stdio.h>

// Declaration of an external variable
extern int extVar;

void printExtVar(); // Declaration of function that uses extVar

int main() {
    printExtVar(); // Use extVar from another file
    return 0;
}
```

```c
// helper.c File
#include <stdio.h>

// Definition and initialization of external variable
int extVar = 42;

void printExtVar() {
    printf("External Variable is: %d\n", extVar);
}
```

1. External Linkage: extern variables are defined in another file or outside of any function, allowing them to be accessed across multiple files.
2. Global Access: Enables variables to be shared between different program files.
3. Declaration vs. Definition: An extern declaration in one file references the variable's memory location defined in another file without allocating space for it.
4. No Initialization: Typically, extern variables are not initialized with the declaration.
5. Use Case: Useful for global variables that need to be accessed by multiple functions across different files.

8.8 Usage of Storage Classes

Storage Classes	Storage	Default value	Scope	Life
Automatic	Memory	Garbage value	Local to the block in which variable is defined	Till the control remains within the block in which the variable is defined.
Register	CPU registers	Garbage value	Local to the block in which variable is defined	Till the control remains within the block in which the variable is defined.
Static	Memory	Zero	Local to the block in which variable is defined	Value of the variable persists between different function calls.
Extern	Memory	Zero	Global	As long as the programs execution doesn't come to an end.

8.8 Usage of Storage Classes

1. auto
   - Default for local variables within functions.
   - Use when you need a variable that is only accessible within its block scope and its lifetime is limited to that block.
2. register
   - For variables that are heavily used, such as loop counters, where you want to hint the compiler to optimize for speed.
   - Keep in mind the compiler might ignore the hint.
3. static
   - For local variables within functions that you want to retain their value between calls to that function.
   - At file scope, to limit the visibility of a global variable to the file it's declared in, enhancing encapsulation.
4. extern
   - When you need to access a variable defined in another file or to declare a global variable across multiple files.
   - Use for shared data while keeping the definition in one place to avoid multiple definition errors.

Revision

1. Long
2. Signed and Unsigned
3. What are Storage Classes
4. Automatic
5. Register
6. Static
7. External
8. Usage of Storage Classes

CHALLENGE

63. Create a program that converts a large number of kilometers to miles, using long or long long to store the distance.
64. Write a program to demonstrate the difference in range between long and long long by calculating the factorial of 20.
65. Write a C program that initializes an unsigned int to its maximum possible value and an int to a negative number. Add 1 to both, and print the results to show how the unsigned int wraps around to 0, whereas the int remains negative due to overflow

KG Coding

Some Other One shot Video Links:

- Complete Java
- Complete HTML & CSS
- Complete JavaScript
- Complete React and Redux
- One shot University Exam Series

http://www.kgcoding.in/

Our YouTube Channels

KG Coding Android App

KG Coding

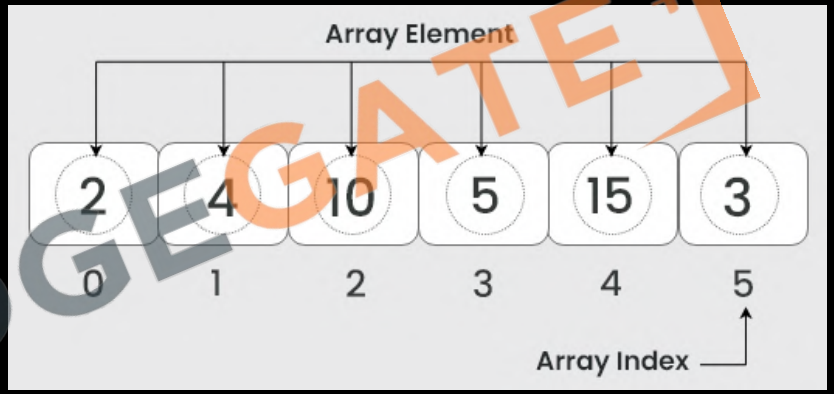
Knowledge GATE

KG Placement Prep

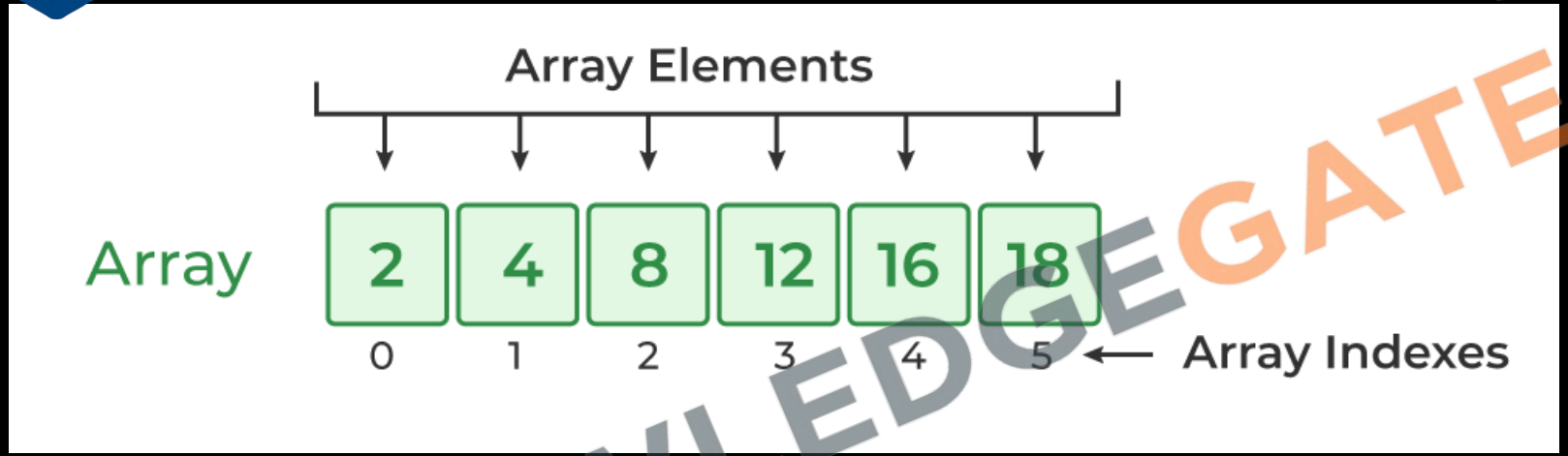
Sanchit Socket

9 Arrays

1. Need of Array
2. Array Declaration
3. Accessing Array Elements
4. Array Initialization
5. Array Traversal
6. Bounds Checking
7. Array as Function Arguments
8. Pointer Arithmetic
9. Pointers and Arrays
10. Two-Dimensional Array
11. Multi-Dimensional Array

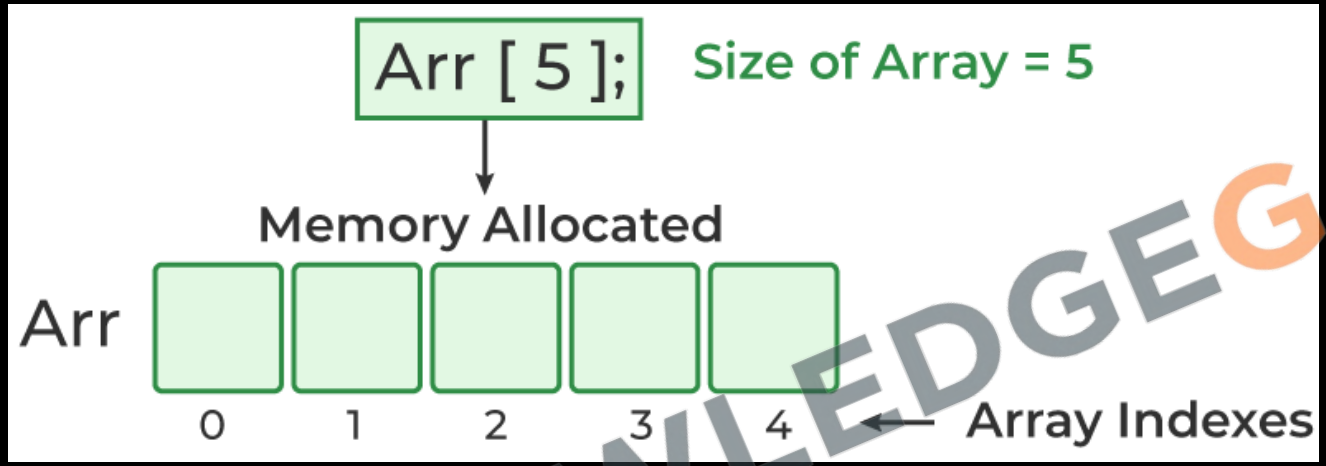


9.1 Need of Array



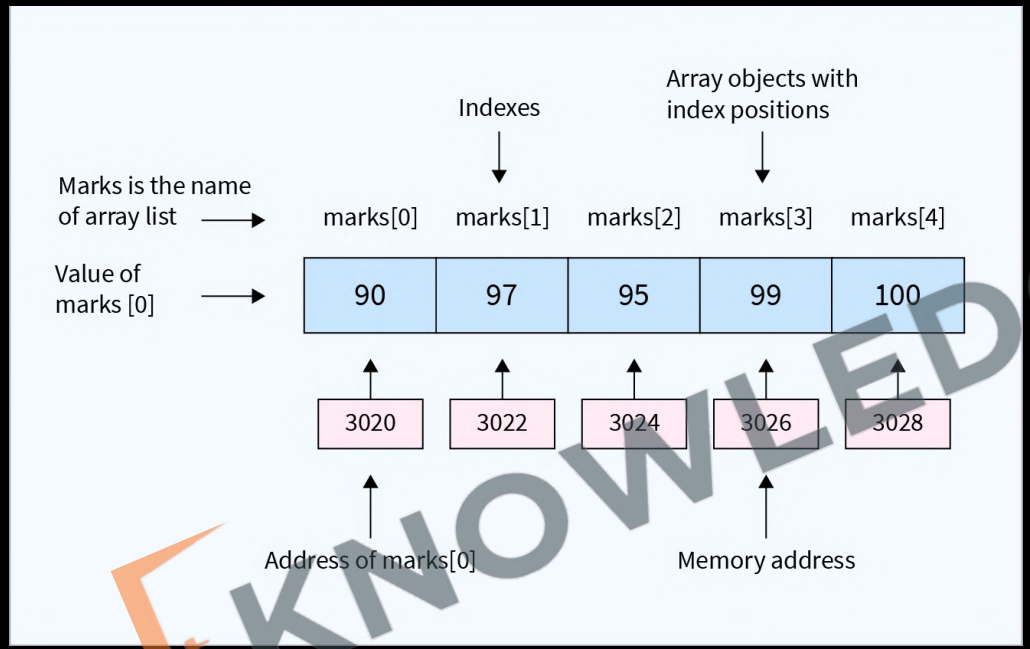
1. An Array is just a list of values.
2. Index: Starts with 0.
3. Arrays are used for storing multiple values in a single variable.

9.2 Array Declaration



1. Syntax: Declared by specifying the type, name, and size in brackets, e.g., int arr[10];.
2. Fixed Size: Size must be known at compile time or be a constant expression.
3. Zero-Based Indexing: Arrays are indexed starting from 0.
4. Storage: Contiguous block of memory.
5. Type Uniformity: All elements must be of the same type.

9.3 Accessing Array Elements



1. Use Indexes: Access with arr[index].
2. Starts at 0: First element is arr[0].
3. For Loops: Iterate with loops.
4. No Bounds Check: Accessing outside the array size is unsafe.

9.3 Accessing Array Elements

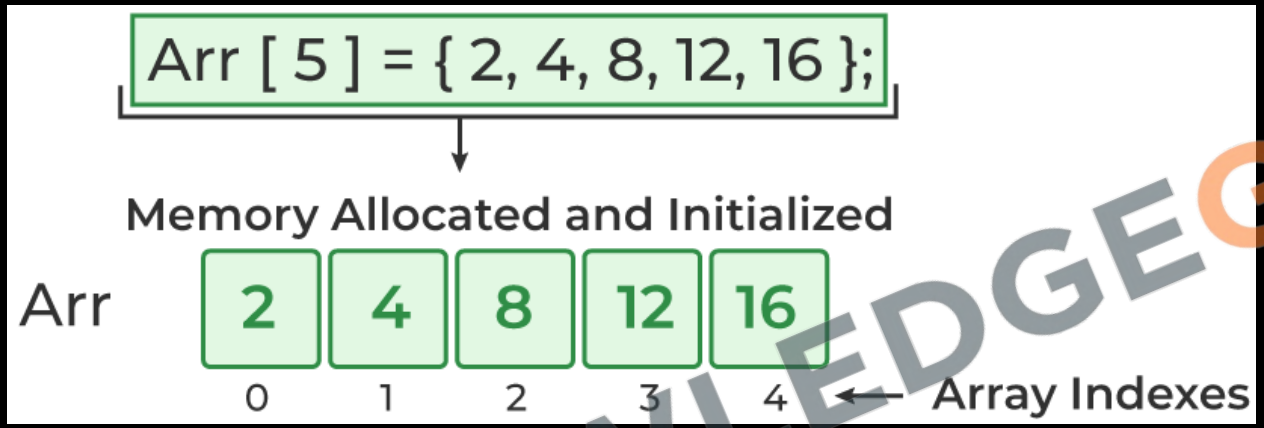
(Array Length)

Array length = Array's Last Index + 1

Array's last index = 7

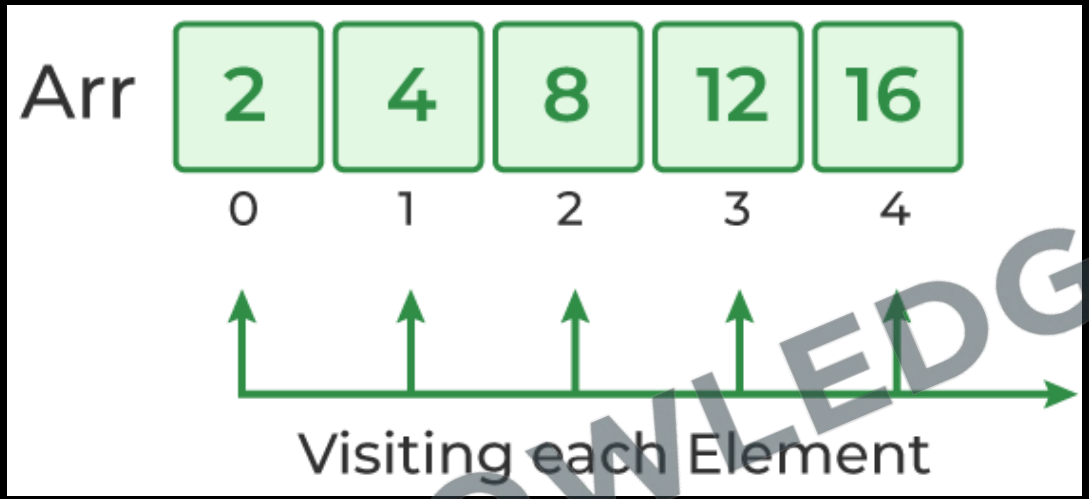
Arraylength = 7 + 1 = 8

9.4 Array Initialization



1. Direct Assignment: Initialize with values, e.g.,
   `int nums[] = {1, 2, 3};`.
2. Auto Zero-fill: Unspecified elements default to zero, e.g.,
   `int arr[5] = {1};`.
3. Zero Array: Initialize all to zero with `int arr[5] = {0};` or empty braces.
4. Designated: Set specific elements, e.g.,
   `int arr[5] = {[2] = 5};`.

9.5 Array Traversal




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

    printf("Array elements:\n");
    for (int i = 0; i < arr_size; i++) {
        printf("%d ", arr[i]);
    }

    return 0;
}
```


1. Orderly Visit: Go through elements from start to end.
2. Loops: Use for or while for access.
3. Indexes: Start at 0, end at size - 1.
4. Modify or Read: Perform operations on elements.
5. Pointer Option: Increment pointers to navigate.

9.6 Bounds Checking

```c
#include <stdio.h>

int main() {
    int num[40];
    for (int i = 0; i < 100; i++) {
        num[i] = i;
    }
    return 0;
}
```

```
[Running] cd "/Users/prashantjain/Desktop/test/
/bin/sh: line 1: 61994 Abort trap: 6

[Done] exited with code=134 in 0.135 seconds
```

1. C Doesn't Auto-check: No automatic bounds verification on arrays.
2. Programmer's Duty: Ensure indices are within array limits.
3. Risks: Out-of-bounds access can lead to crashes or security risks.
4. Validate Indices: Always check indices against array size.

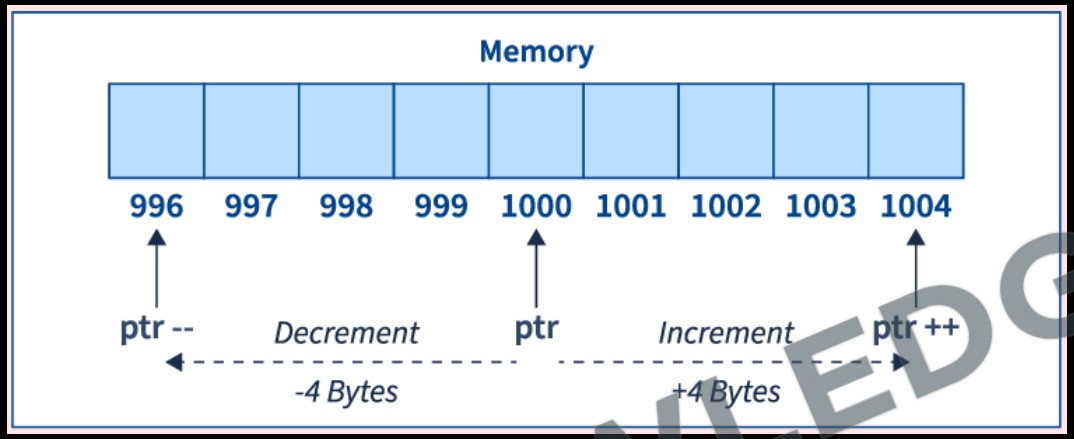
9.7 Array as Function Arguments

```c
// Function prototype
int sumArray(int arr[], int size);

int main() {
    int myArray[] = {1, 2, 3, 4, 5};
    int size = sizeof(myArray) / sizeof(myArray[0]);
    // Pass array and size to the function
    int total = sumArray(myArray, size);
    printf("The sum of the array elements is: %d\n", total);
    return 0;
}

// Function to calculate the sum of an array's elements
int sumArray(int arr[], int size) {
    int sum = 0;
    for (int i = 0; i < size; i++) {
        sum += arr[i];
    }
    return sum; // Return the sum
}
```

```
[Running] cd "/Users/prashantjain/Desktop/test/"
The sum of the array elements is: 15

[Done] exited with code=0 in 0.406 seconds
```

1. Arrays to Pointers: Arrays become pointers when passed to functions.
2. Call By Reference: Changes in functions affect the original array.
3. Include Size: Pass array size as an extra argument.
4. Specify Type: Declare element type in the function parameters.
5. Efficient: No full array copy, just the pointer is passed.

9.8 Pointer Arithmetic



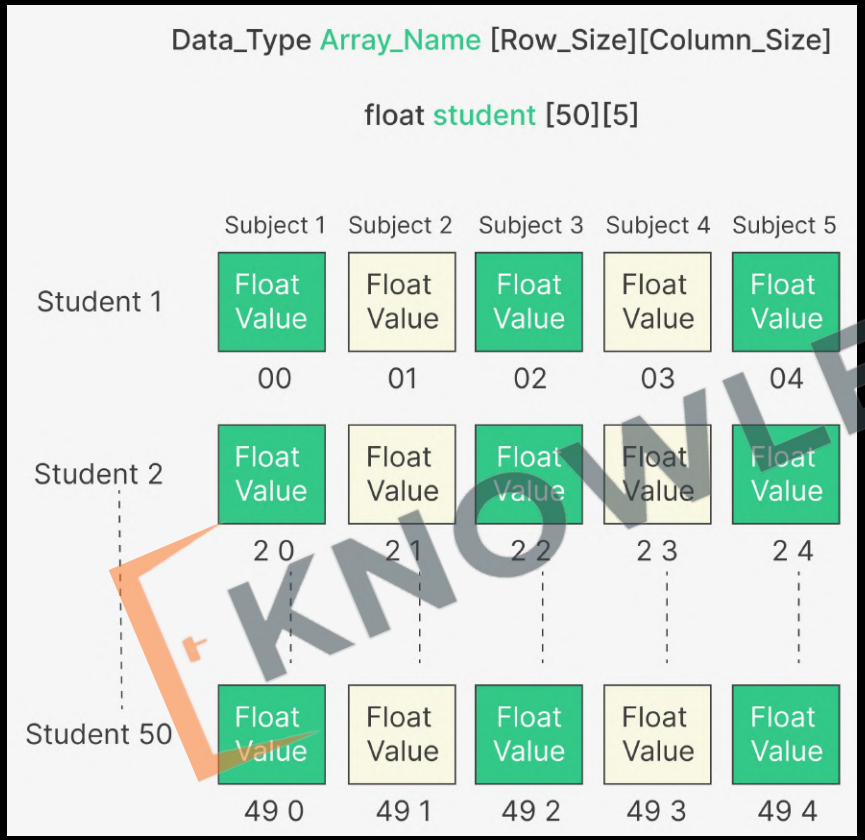
*(arr + i) equals arr[i]

1. Memory Increment/Decrement: ptr++ or ptr-- moves the pointer to the next or previous memory location based on the type's size.
2. Addition/Subtraction: Adding or subtracting an integer n from a pointer moves it n elements forward or backward.
3. Difference: Subtracting one pointer from another gives the number of elements between them.

9.9 Pointers and Arrays

```c
// Function prototype
int sumArray(int *arr, int size);

int main() {
    int myArray[] = {1, 2, 3, 4, 5};
    int size = sizeof(myArray) / sizeof(myArray[0]);
    // Array decays into a pointer when passed
    int total = sumArray(myArray, size);

    printf("The sum of the array elements is: %d\n", total);
    return 0;
}

int sumArray(int *arr, int size) {
    int sum = 0;
    for (int i = 0; i < size; i++) {
        // Access elements via pointer arithmetic
        sum += arr[i];
    }
    return sum; // Return the sum
}
```

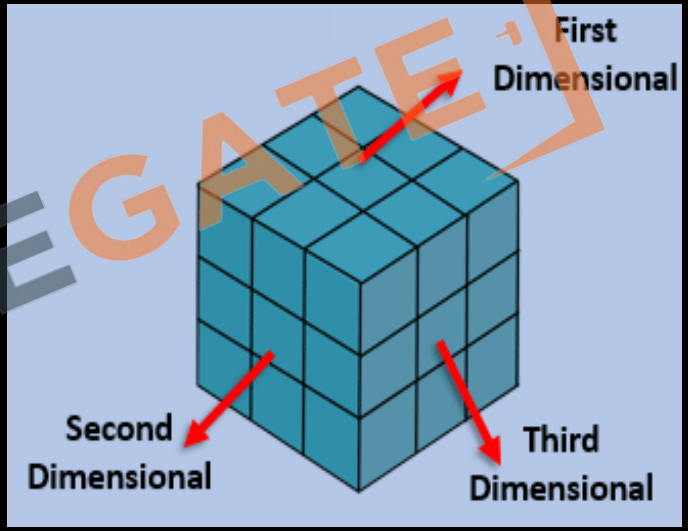
9.10 Two-Dimensional Array



1. Structure: Think of as an array of arrays, often used to represent matrices or grids.
2. Declaration: int arr[3][4]; for 3 rows and 4 columns.
3. Initialization: int arr[2][2] = {{1, 2}, {3, 4}};.
4. Accessing Elements: Use row and column indices, e.g., arr[0][1] for the first row's second element.
5. Memory Layout: Stored in row-major order, meaning rows are stored in contiguous memory locations.

9.11 Multi-Dimensional Array

1. Generalization: Extend two-dimensional arrays to more dimensions, like 3D for cubes or higher.
2. Declaration: Specify each dimension, e.g., int arr[2][3][4] for a 3D array.
3. Initialization: Nested braces for dimensions, e.g., int arr[2][2][2] = {{{1, 2}, {3, 4}}, {{5, 6}, {7, 8}}};.
4. Element Access: Use indices for each dimension, e.g., arr[1][0][1] for a 3D array element.
5. Row-major Order: Memory layout continues with elements stored in contiguous locations, dimension by dimension.



Revision

1. Need of Array
2. Array Declaration
3. Accessing Array Elements
4. Array Initialization
5. Array Traversal
6. Bounds Checking
7. Array as Function Arguments
8. Pointer Arithmetic
9. Pointers and Arrays
10. Two-Dimensional Array
11. Multi-Dimensional Array

CHALLENGE

66. Create a program to find the sum and average of all elements in an array.
67. Create a program to find number of occurrences of an element in an array.
68. Create a program to find the maximum and minimum element in an array.
69. Create a program to check if the given array is sorted.
70. Create a program to return a new array deleting a specific element.
71. Create a program to reverse an array.
72. Create a program to check is the array is palindrome or not.
73. Write a function that uses pointer arithmetic to copy an array of char into another.
74. Create a program to merge two sorted arrays.
75. Create a program to search an element in a 2-D array.
76. Create a program to do sum and average of all elements in a 2-array.
77. Create a program to find the sum of two diagonal elements.

KG Coding

Some Other One shot Video Links:

- Complete Java
- Complete HTML & CSS
- Complete JavaScript
- Complete React and Redux
- One shot University Exam Series

http://www.kgcoding.in/

Our YouTube Channels

KG Coding Android App

KG Coding

Knowledge GATE

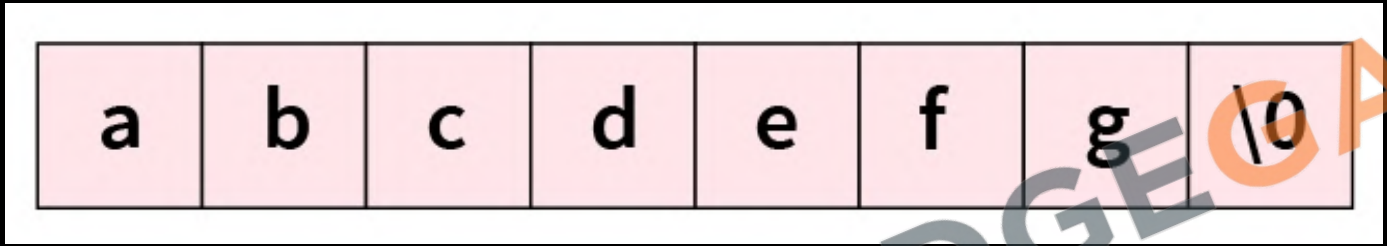
KG Placement Prep

Sanchit Socket

10 Strings

1. What is a String
2. String Memory Allocation
3. String Initialization
4. Format Specifiers
5. fgets and puts functions
6. Pointers and Strings
7. String.h (strlen, strcpy, strcat, strcmp)
8. 2-D Array of Characters

10.1 What is a String



1. Character Array: A string is a sequence of characters terminated by a null character ('\0').
2. Null Termination: The null character ('\0') marks the end of the string, not included in its length.
3. String Literals: Defined in double quotes, e.g., "Hello, World!".
4. Mutable: Strings can be modified when stored in a character array.

10.2 String Memory Allocation

```
char str[6] = "Hello";
```

index	0	1	2	3	4	5
value	H	e	l	l	o	\0
address	1000	1001	1002	1003	1004	1005

1. Memory Allocation: Fixed size at compile time, including the null terminator.
2. Contiguous Memory: Strings are stored contiguously like Arrays.

10.3 String Initialization

```
char c[] = "abcd";
      OR,
char c[50] = "abcd";
      OR,
char c[] = {'a', 'b', 'c', 'd', '\0'};
      OR,
char c[5] = {'a', 'b', 'c', 'd', '\0'};
```

a	b	c	d	\0

10.4 Format Specifiers

```c
int main()
{
    // Allocate a character array
    char str[50];

    printf("Enter a string: ");
    // Read a string with a safety limit
    scanf("%49s", str);
    // Output the entered string
    printf("You entered: %s\n", str);

    return 0;
}
```

```
Enter a string: what is this
You entered: what
```

1. Output: %s is used with printf to output a string.
2. Input: %s is used with scanf to read a string into a character array. It stops reading upon encountering whitespace, a newline, or EOF.
3. No & with scanf: Unlike other data types, when using %s with scanf, you don't prefix the variable name with & because a string name acts as a pointer.
4. Safety: To prevent buffer overflow with scanf, specify a maximum field width, e.g., %9s for a char array of size 10.

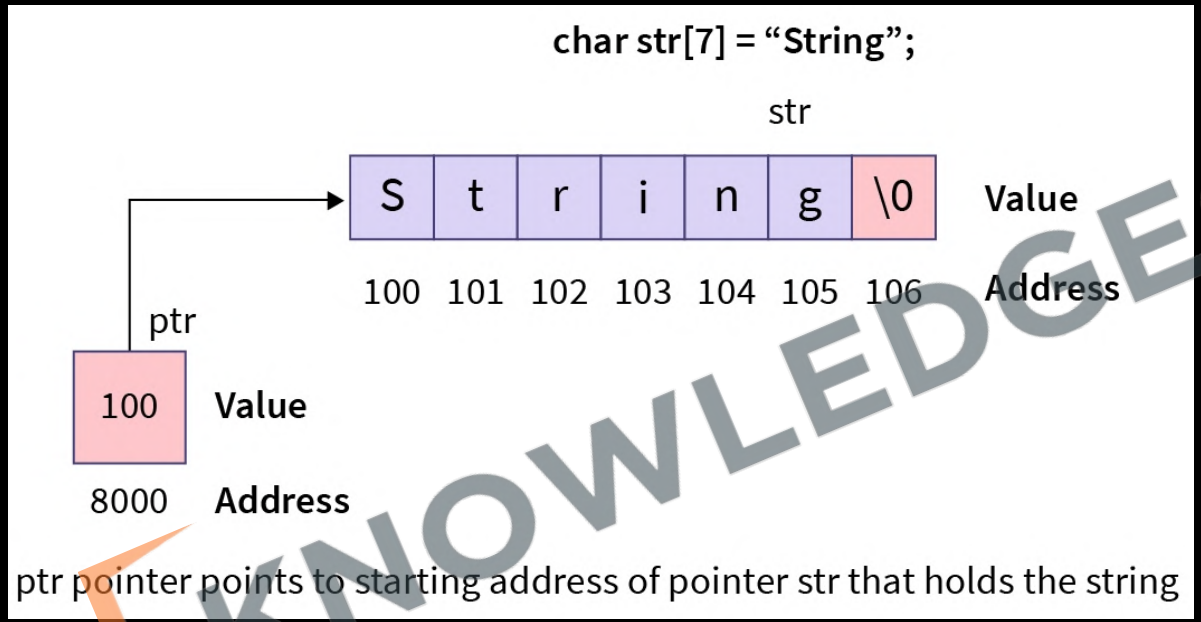
10.5 fgets and puts functions

```c
int main() {
    char str[100];

    printf("Enter a string using gets: ");
    gets(str); // UNSAFE: Do not use in production code

    printf("You entered (gets): ");
    puts(str); // Outputs the string followed by a newline

    printf("Enter a string using fgets: ");
    fgets(str, sizeof(str), stdin); // Safe alternative to gets()

    printf("You entered (fgets): ");
    puts(str); // Outputs the string followed by a newline

    return 0;
}
```

10.5 fgets and puts functions

1. fgets():
   - Safe Input: Reads a string from a file or stdin safely into a buffer.
   - Limits Length: Takes the maximum size to read, preventing buffer overflow.
   - Includes Newline: Can include the newline character (\n) if it fits in the buffer.
2. puts():
   - Simple Output: Writes a string to stdout and appends a newline automatically.
   - No Format Specifiers: Directly prints the string without formatting options.
3. gets() (Not Recommended):
   - Unsafe Input: Reads a string from stdin until a newline or EOF without size limits.
   - Buffer Overflow: Prone to overflow, leading to security vulnerabilities.
   - Deprecated: Removed from the C11 standard due to its risks.

10.6 Pointers and Strings



A string can be represented as a pointer to its first character. For example, char *str = "Hello"; makes str a pointer to the first character of the string literal "Hello".

10.6 Pointers and Strings

```c
int main() {
    // Character array - modifiable
    char arrStr[] = "Hello";
    arrStr[0] = 'J'; // Modifying the first character
    printf("Modified array string: %s\n", arrStr);

    // Character pointer to string literal - not modifiable
    char *ptrStr = "Hello";
    // Attempting to modify the first character
    ptrStr[0] = 'J'; // This will lead to undefined behavior

    printf("Pointer string (attempted modification): %s\n", ptrStr);

    return 0;
}
```


1. Immutable String Literals: String literals assigned to character pointers are stored in read-only memory, making them immutable. Attempting to modify them, e.g., str[0] = 'M';, leads to undefined behavior.
2. Array vs. Pointer: While an array name is a constant pointer to its first element, a character array allows modification of its elements.

10.7 String.h
(strlen)

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

int main() {
    char str1[] = "Hello";
    // strlen usage
    int len = strlen(str1);
    printf("Length of '%s' is %d\n", str1, len);
    return 0;
}
```

```
Length of 'Hello' is 5
```

Returns the length of a string, not including the terminating null byte ('\0')

10.7 String.h
(strcpy)

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

int main() {
    char str1[] = "Hello";
    char str2[20];
    // strcpy usage
    strcpy(str2, str1);
    printf("Copied string: %s\n", str2);
    return 0;
}
```

```
Copied string: Hello
```

Copies the string pointed to by the source, including the terminating null byte, to the destination

10.7 String.h
(strcat)

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

int main() {
    char str1[20] = "Hello, ";
    char str2[] = "World!";
    // strcat usage
    strcat(str1, str2);
    printf("Concatenated string: %s\n", str1);
    return 0;
}
```

```
Concatenated string: Hello, World!
```

Appends the source string to the destination string, overwriting the terminating null byte at the end of destination, and then adds a terminating null byte.

10.7 String.h
(strcmp)

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

int main() {
    // Comparing "apple" with "banana"
    int comparisonResult = strcmp("apple", "banana");
    printf("Comparing 'apple' with 'banana': %d\n", comparisonResult);

    // Comparing "cherry" with "banana"
    comparisonResult = strcmp("cherry", "banana");
    printf("Comparing 'cherry' with 'banana': %d\n", comparisonResult);

    // Comparing "date" with "date"
    comparisonResult = strcmp("date", "date");
    printf("Comparing 'date' with 'date': %d\n", comparisonResult);

    return 0;
}
```

```
Comparing 'apple' with 'banana': -1
Comparing 'cherry' with 'banana': 1
Comparing 'date' with 'date': 0
```

Compares two strings lexicographically and returns an integer based on the comparison.

10.8 2-D array of Characters

```c
int main() {
    // Declare and initialize a 2D array of characters
    // Each inner array represents a string
    char names[3][20] = {
        "Alice",
        "Bob",
        "Charlie"
    };

    // Modify a string
    names[1][0] = 'R'; // Changing "Bob" to "Rob"

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

    // Accessing and printing a specific character
    // Should print 'h'
    printf("Second character of the third name: %c\n", names[2][1]);

    return 0;
}
```

```
Alice
Rob
Charlie
Second character of the third name: h
```

10.8 2-D array of Characters

1. Matrix of Characters: In a 2D array of characters each row represents a string.
2. String Array: Often used to store multiple strings of the same maximum length.
3. Null Termination: Each string in the array must end with a null character '\0'.
4. Fixed Width: All strings have a fixed maximum width, defined by the column size of the 2D array.
5. Row Access: Access a specific string (row) with single indexing,
   e.g., charArray[0] accesses the first string.
6. Character Access: Access individual characters with double indexing,
   e.g., charArray[1][2] accesses the 3rd character of the 2nd string.
7. Initialization: Can be initialized row by row using string literals,
   e.g., char charArray[3][4] = {"abc", "def", "ghi"};.

Revision

1. What is a String
2. String Memory Allocation
3. String Initialization
4. Format Specifiers
5. fgets and puts functions
6. Pointers and Strings
7. String.h (strlen, strcpy, strcat, strcmp)
8. 2-D Array of Characters

CHALLENGE

78. Read a line of text from the user using fgets and then print it using puts.
79. Use printf with format specifiers to format and print a date string (day, month, year).
80. Write a program to convert a input string to uppercase.
81. Create a simple text-based user login system that compares a stored password string using strcmp.
82. Use a 2-D character array to store and display a tic-tac-toe board.
83. Write a function that takes a string and reverses it in place.
84. Implement a trim function that removes leading and trailing spaces from a string.
85. Create a program that checks if a given string is a palindrome (the same forwards and backwards) and outputs the result.
86. Create a program using do-while to find password checker until a valid password is entered.
87. Create a program using break to read inputs from the user in a loop and break the loop if a specific keyword (like "exit") is entered.

KG Coding

Some Other One shot Video Links:

- Complete Java
- Complete HTML & CSS
- Complete JavaScript
- Complete React and Redux
- One shot University Exam Series

http://www.kgcoding.in/

Our YouTube Channels

KG Coding Android App

KG Coding

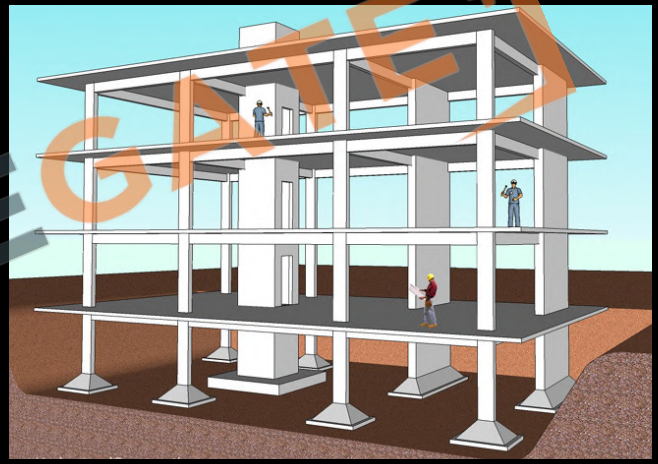
Knowledge GATE

KG Placement Prep

Sanchit Socket

11 Structures

1. Why use Structure
2. Structure Declaration
3. Accessing Structure Elements
4. Structure Memory Allocation
5. Structure Initialization
6. Array of Structures
7. Structure Pointers
8. Arrow -> Operator
9. Structure as Function Arguments
10. Typedef keyword



11.1 Why use Structure

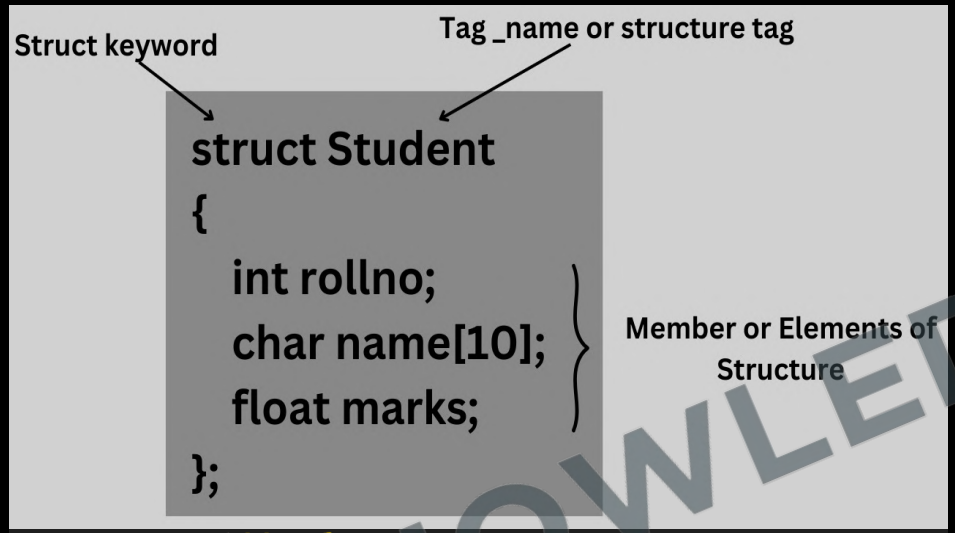
```c
#include <stdio.h>

int main() {
    int rollNumbers[5] = {1, 2, 3, 4, 5};
    char names[5][10] = {"Student1", "Student2",
     "Student3", "Student4", "Student5"};
    float marks[5] = {90.5, 85.0, 92.5, 88.0, 95.0};

    for
        int printf(const char *__restrict__, ...)
        printf("Roll Number: %d\n", rollNumbers[i]);
        printf("Name: %s\n", names[i]);
        printf("Marks: %.2f\n\n", marks[i]);
    }

    return 0;
}
```

```
Roll Number: 1
Name: Student1
Marks: 90.50

Roll Number: 2
Name: Student2
Marks: 85.00

Roll Number: 3
Name: Student3
Marks: 92.50

Roll Number: 4
Name: Student4
Marks: 88.00

Roll Number: 5
Name: Student5
Marks: 95.00
```

11.1 Why use Structure

1. Grouping Data: Structures allow related data items of different types to be grouped together under a single name.
2. Modelling Real-world Entities: Enable the representation of complex entities (e.g., a student, a point in space) more naturally.
3. Ease of Handling: Simplify passing multiple data items as a single argument to functions.
4. Data Encapsulation: Encapsulate related data, enhancing code readability and maintainability.
5. Type Definition: Allow the creation of new data types tailored to specific needs.

11.2 Structure Declaration




```c
int main() {
    // creating a student variable
    struct Student student;
}
```

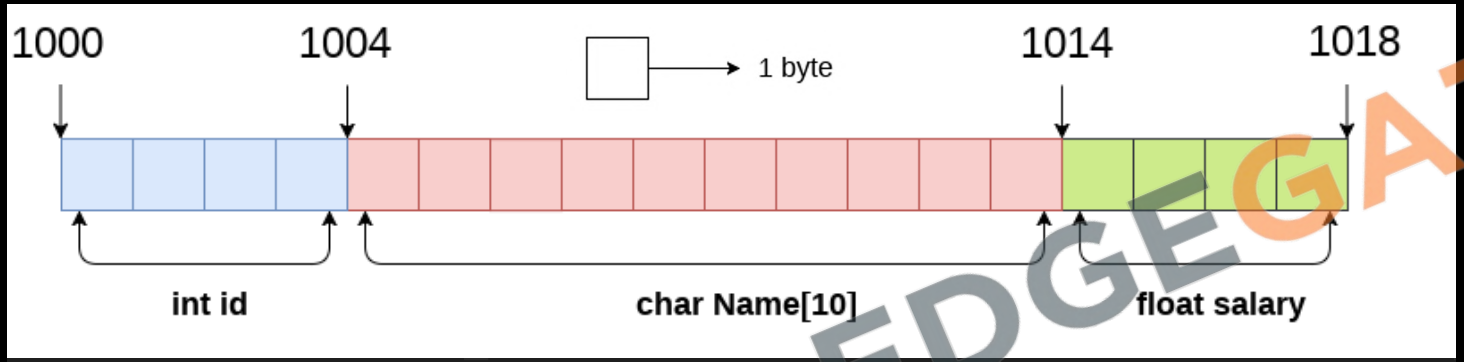
1. Keyword: Use the struct keyword to define a structure.
2. Name: Optionally give a name to the structure type.
3. Members: Enclose member declarations in braces {}.
4. Semicolon: End the structure definition with a semicolon ;.

11.3 Accessing Structure Elements

```c
int main() {
    struct Student ram;
    // Input for Ram
    printf("Enter details for Ram:\n");
    printf("Roll No: ");
    scanf("%d", &ram.rollno);
    printf("Marks: ");
    scanf("%f", &ram.marks);
    // Output
    printf("\nStudent Details:\n");
    printf("Ram - Roll No: %d, Marks: %.2f\n",
     ram.rollno, ram.marks);
    return 0;
}
```

1. Memory Dot Operator: student.rollno for direct access.
2. Nested Access: school.class.student.rollno for nested structures.

11.4 Structure Memory Allocation




```c
struct Student {
    int rollno;
    char name[50];
    float marks;
};
```

```
sizeof (student) = 4 + 10 + 4 = 18 bytes
where;
sizeof (int) = 4 byte
sizeof (char) = 1 byte
sizeof (float) = 4 byte
```


1. Contiguous Block: Structures occupy contiguous memory spaces.
2. Padding: Compilers may add padding for alignment, affecting size.
3. Size: Use sizeof to get the structure's size, padding included.

11.5 Structure Initialization

```c
int main() {
    // Direct Initialization
    struct Student s1 = {1, "Alice", 85.5};
    // Designated Initializers (C99)
    struct Student s2 = {.rollno = 2, .name = "Bob", .marks = 90.2};
    // Zero Initialization
    struct Student s3 = {0};
    // Copy Initialization
    struct Student s4 = s1;

    printf("Student1: %d, %s, %.1f\n", s1.rollno, s1.name, s1.marks);
    printf("Student2: %d, %s, %.1f\n", s2.rollno, s2.name, s2.marks);
    printf("Student3: %d, %s, %.1f\n", s3.rollno, s3.name, s3.marks);
    printf("Student4: %d, %s, %.1f\n", s4.rollno, s4.name, s4.marks);
    return 0;
}
```

```
Student1: 1, Alice, 85.5
Student2: 2, Bob, 90.2
Student3: 0, , 0.0
Student4: 1, Alice, 85.5
```

1. Direct Initialization: Use curly braces {} directly after declaration, specifying values in order.
2. Designated Initializers: C99 allows initializing specific fields by name, enhancing readability and flexibility.
3. Zero Initialization: Assigning {0} initializes all members to zero or null equivalents.
4. Copy Initialization: Initialize one structure with another of the same type.

11.6 Array of Structures

```c
int main() {
    // Initializing an array of Student structures
    struct Student students[5] = {
        {1, "Student1", 90.5},
        {2, "Student2", 85.0},
        {3, "Student3", 92.5},
        {4, "Student4", 88.0},
        {5, "Student5", 95.0}
    };

    // Loop to print each student's data
    for (int i = 0; i < 5; i++) {
        printf("Roll Number: %d\n", students[i].rollno);
        printf("Name: %s\n", students[i].name);
        printf("Marks: %.2f\n\n", students[i].marks);
    }
    return 0;
}
```

```
Roll Number: 1
Name: Student1
Marks: 90.50

Roll Number: 2
Name: Student2
Marks: 85.00

Roll Number: 3
Name: Student3
Marks: 92.50

Roll Number: 4
Name: Student4
Marks: 88.00

Roll Number: 5
Name: Student5
Marks: 95.00
```

11.7 Structure Pointer

```c
// Declare a Student structure variable
struct Student student = {1, "Student1", 90.5};
// Pointer to the structure
struct Student *studentPtr = &student;
```


1. Definition: Structure pointers are pointers that point to structure variables.
2. Pointer Arithmetic: For structure arrays same pointer arithmetic applies.

11.8 Arrow -> Operator

```c
int main() {
    // Declare a Student structure variable
    struct Student student;
    struct Student *studentPtr = &student; // Pointer to the
    structure

    // Assign values to the structure using the pointer
    studentPtr->rollno = 1;
    // Use strcpy to copy string
    strcpy(studentPtr->name, "Sita");
    studentPtr->marks = 88.5;

    // Access and print structure members using the pointer
    printf("Student Details:\n");
    printf("Roll Number: %d\n", studentPtr->rollno);
    printf("Name: %s\n", studentPtr->name);
    printf("Marks: %.2f\n", studentPtr->marks);

    return 0;
}
```

1. Purpose: Used to access members of a structure or union through a pointer.
2. Syntax: pointer->member accesses the member of the structure that pointer points to.
3. Alternative: Equivalent to (*pointer).member, but more concise.
4. Use Case: Simplifies accessing structure members without dereferencing the pointer explicitly.
5. Common in Linked Lists: Frequently used in data structures like linked lists for navigating through nodes.

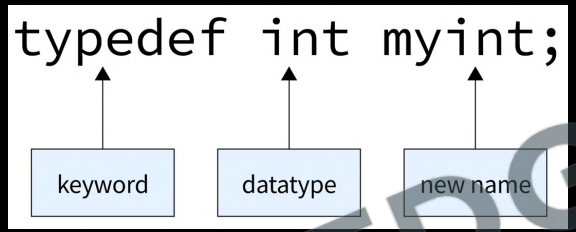
11.9 Structure as Function Arguments

```c
// Function prototype
void printStudentPointer(struct Student *s);
void printStudent(struct Student s);

int main() {
    struct Student student = {1, "Sita", 88.5};
    struct Student *studentPtr = &student;
    printStudentPointer(studentPtr);
    printStudent(student);
    return 0;
}

void printStudentPointer(struct Student *s) {
    printf("Student Details:\n");
    printf("Roll Number: %d\n", s->rollno);
    printf("Name: %s\n", s->name);
    printf("Marks: %.2f\n", s->marks);
}

void printStudent(struct Student s) {
    printf("Student Details:\n");
    printf("Roll Number: %d\n", s.rollno);
    printf("Name: %s\n", s.name);
    printf("Marks: %.2f\n", s.marks);
}
```

1. Memory: Direct passing copies the whole structure; passing by address is more efficient.
2. Modification: Changes in the function affect the original only when passed by address.
3. Performance: Passing by address is faster, especially for large structures.
4. Simplicity: Direct passing is simpler but less flexible; passing by address allows modifications.

11.10 Typedef keyword

```c
struct Student {
    int rollno;
    char name[50];
    float marks;
};

int main() {
    struct Student ram;
    struct Student shyam;
    return 0;
}
```




```c
int main() {
    typedef struct Student S;
    S ram;
    S shyam;
    return 0;
}
```

```c
typedef struct {
    int rollno;
    char name[50];
    float marks;
} Student ;
```


1. Alias Creation: typedef creates aliases for existing types, making them easier to work with.
2. Structure Simplification: Often used to simplify the syntax for structures.
3. Readability: Enhances code readability and maintainability by using more meaningful names.
4. Portability: Facilitates portability across different systems by abstracting type specifics.

Revision

1. Why use Structure
2. Structure Declaration
3. Accessing Structure Elements
4. Structure Memory Allocation
5. Structure Initialization
6. Array of Structures
7. Structure Pointers
8. Arrow -> Operator
9. Structure as Function Arguments
10. Typedef keyword

CHALLENGE

88. Create a program where you need to store and process data for a Book with attributes like title, author, and price, demonstrating why a structure is more suitable than separate variables.
89. Initialize an array of Book structures with different data for each book using designated initializers.
90. Define a Car structure with fields for make, model, year, and color.
91. Pass a Car structure to a function that prints out a description of the car in one complete sentence.
92. Write a function that accepts a pointer to a Student structure with fields for id, name, year, gpa and modifies its grades.
93. Write a function where the Student structure also has books they have borrowed inside, showing nested structure usage.

KG Coding

Some Other One shot Video Links:

- Complete Java
- Complete HTML & CSS
- Complete JavaScript
- Complete React and Redux
- One shot University Exam Series

http://www.kgcoding.in/

Our YouTube Channels

KG Coding Android App

KG Coding

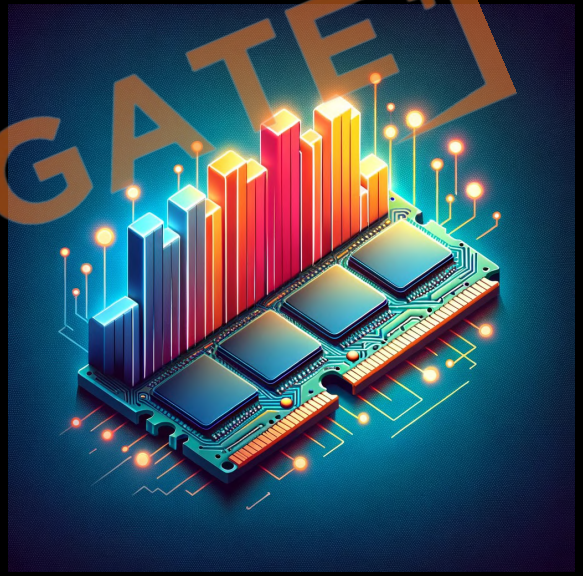
Knowledge GATE

KG Placement Prep

Sanchit Socket

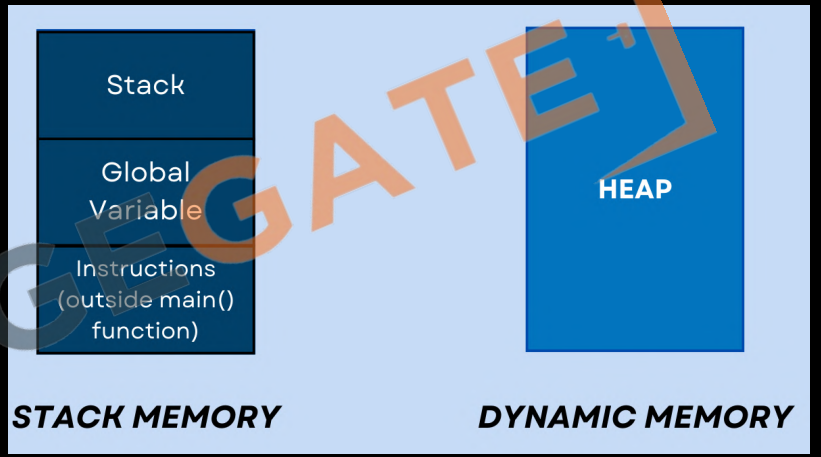
12 Dynamic Memory Allocation

1. What is Dynamic Memory Allocation
2. Using malloc
3. Using calloc
4. Using free
5. Using realloc

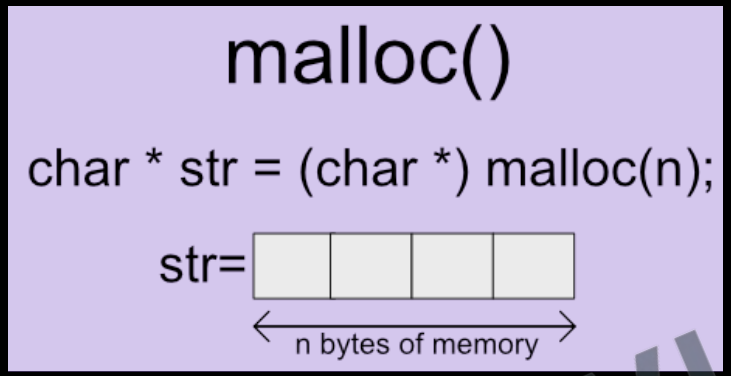


12.1 What is Dynamic Memory Allocation

1. Flexibility: Allocate memory as needed at runtime, accommodating variable data size requirements.
2. Efficiency: Use only the memory necessary, releasing it when no longer needed, which can be more efficient than static allocation.
3. Data Persistence: Memory allocated dynamically persists beyond the scope it was created in, until it is explicitly freed.
4. Large Data: Enables handling of large data sets that might exceed the stack size, avoiding stack overflow.
5. Data Structures: Essential for complex data structures like linked lists, trees, and graphs where the size is not known at compile time.



12.2 Using Malloc



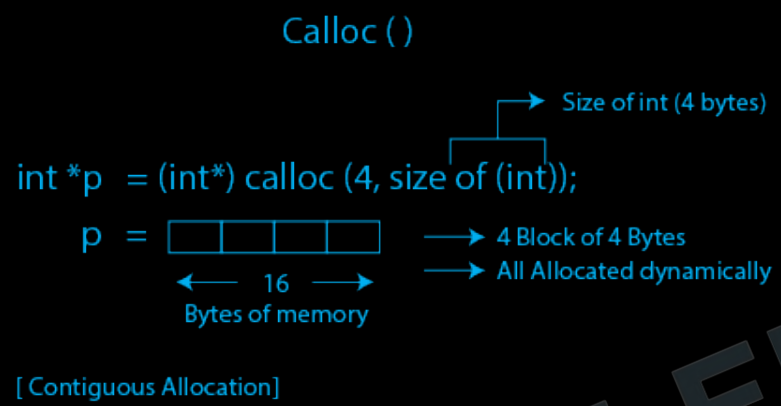
1. Include Header: Use #include <stdlib.h> to access malloc.
2. Size Argument: Pass the size in bytes of the memory you need to allocate.
3. Pointer Typecasting: Cast the returned void* pointer to the appropriate type.
4. Check for NULL: Always check if malloc returns NULL, indicating allocation failure.
5. Memory Initialization: malloc does not initialize memory; it's raw and may contain garbage values.

12.2 Using Malloc

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

int main() {
    int *arr;
    int n;
    int sum = 0; // Variable to store the sum

    printf("Enter the number of elements: ");
    scanf("%d", &n);

    // Dynamically allocate memory using malloc
    arr = (int*)malloc(n * sizeof(int));

    // Check if the memory has been successfully allocated
    if (arr == NULL) {
        printf("Memory allocation failed\n");
        return 1;
    }
```

```c
    // Input elements into the array and sum them up
    printf("Enter %d elements:\n", n);
    for (int i = 0; i < n; i++) {
        printf("Enter element %d: ", i);
        scanf("%d", &arr[i]);
        sum += arr[i]; // Add element to sum
    }

    // Print the sum of the elements
    printf("The sum of the array elements is: %d\n", sum);
    return 0;
}
```

12.3 Using calloc



1. Initialization: calloc initializes all allocated memory to zero.
2. Syntax: Takes two arguments, the number of elements and the size of each element.
3. Usage: Preferred for array allocation when zero-initialization is needed.
4. Memory Overhead: May have slightly more overhead than malloc due to initialization.
5. Return Type: Like malloc, returns a void* pointer that should be cast to the appropriate type.

12.3 Using calloc

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

int main() {
    int *arr;
    int n;
    int sum = 0; // Variable to store the sum

    printf("Enter the number of elements: ");
    scanf("%d", &n);

    // Dynamically allocate memory using calloc
    arr = (int*)calloc(n, sizeof(int));

    // Check if the memory has been successfully allocated
    if (arr == NULL) {
        printf("Memory allocation failed\n");
        return 1;
    }

    // Input elements into the array and sum them up
    printf("Enter %d elements:\n", n);
    for (int i = 0; i < n; i++) {
        printf("Enter element %d: ", i);
        scanf("%d", &arr[i]);
        sum += arr[i]; // Add element to sum
    }

    // Print the sum of the elements
    printf("The sum of the array elements is: %d\n", sum);
    return 0;
}
```

12.4 Using free

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

int main() {
    // Allocate memory for an integer
    int *p = (int*)malloc(sizeof(int));
    if (p == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }

    *p = 10; // Assign a value
    printf("Value: %d\n", *p);

    free(p); // Free the allocated memory
    p = NULL; // Avoid dangling pointer

    return 0;
}
```

1. Memory Release: free deallocates previously allocated memory, making it available for future allocations.
2. Prevent Leaks: Essential for preventing memory leaks in programs that allocate dynamic memory.
3. Pointer Argument: Takes a pointer to the allocated memory block as its argument.
4. Null Safe: Calling free with a NULL pointer is safe and results in no operation.
5. Use Once: After calling free, the pointer should not be used until it points to another allocated memory block or is set to NULL.

12.5 Using realloc

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

int main() {
    // Allocate memory for 2 integers
    int* arr = malloc(2 * sizeof(int));
    arr[0] = 1;
    arr[1] = 2;

    // Resize array to hold 3 integers
    arr = realloc(arr, 3 * sizeof(int));
    arr[2] = 3; // Initialize new element

    // Print new array size elements
    for (int i = 0; i < 3; ++i) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    free(arr); // Free the allocated memory
    return 0;
}
```

1. Resize Memory: realloc is used to resize previously allocated memory without losing the data.
2. New Allocation: If passed a NULL pointer, it behaves like malloc.
3. Freeing: If the new size is 0, it behaves like free, deallocating the memory.
4. Data Preservation: Attempts to preserve the original data, even when moving to a new location.
5. Return Value: Returns a pointer to the newly allocated memory, which may differ from the original pointer.
6. Error Handling: Returns NULL on failure without freeing the original block, so always check the return value before using it.

Revision

1. What is Dynamic Memory Allocation
2. Using malloc
3. Using calloc
4. Using free
5. Using realloc

CHALLENGE

94. Create a program that dynamically allocates memory for a Car structure and then free it at the end of the program.
95. Create a program that uses malloc to dynamically allocate an array for a specified number of float values entered by the user and then stores user-entered numbers into it.
96. Use calloc to allocate an array for a set of char characters representing a sentence, ensuring all the memory is initialized to zero.
97. Allocate memory for a struct representing a Point with x and y coordinates, set some values, and then properly deallocate the memory using free.
98. Create an array using calloc and fill it with random numbers, then use realloc to shrink the array size by half and print the remaining numbers.

KG Coding

Some Other One shot Video Links:

- Complete Java
- Complete HTML & CSS
- Complete JavaScript
- Complete React and Redux
- One shot University Exam Series

http://www.kgcoding.in/

Our YouTube Channels

KG Coding Android App

KG Coding

Knowledge GATE

KG Placement Prep

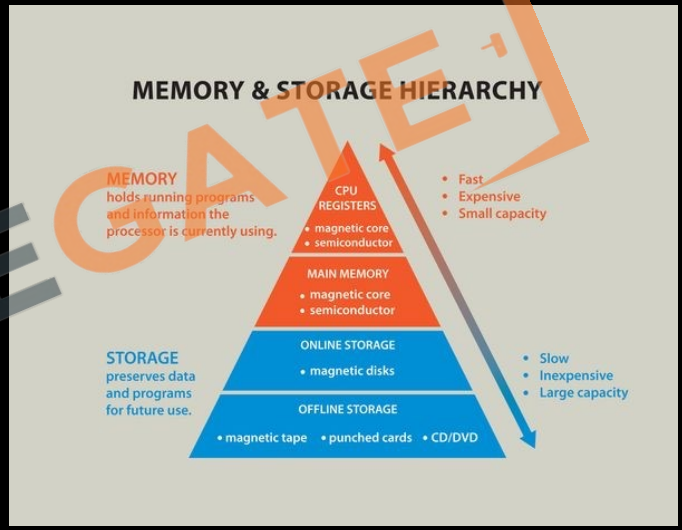
Sanchit Socket

13 File Input/Output

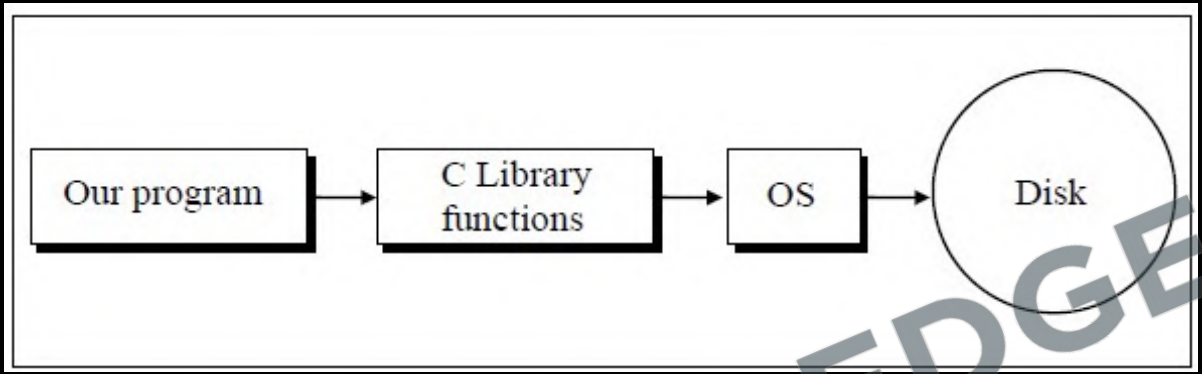
1. Data Organization
2. File Operations
3. Text Files & Binary Files
4. File pointer
5. Open a File
6. Close a File
7. Reading data from File
8. End of File (EOF)
9. Writing data to a File
10. Appending data to a File
11. File Opening Modes

13.1 Data Organization

1. RAM (Random Access Memory): Volatile memory for temporary data storage while a computer is running.
2. ROM (Read-Only Memory): Non-volatile memory used primarily to store firmware.
3. Hard Disk Drives (HDDs): Mechanical, magnetic storage devices for long-term data retention.
4. Solid-State Drives (SSDs): Fast, non-volatile storage devices that use flash memory.
5. Cache Memory: A small, speedy volatile memory in CPUs or between CPU and RAM to store frequent data.
6. Virtual Memory: Part of a hard drive used as RAM for extending physical memory, albeit slower.
7. Flash Memory: Non-volatile memory used in USB drives and memory cards for portability.
8. Optical Storage: Media like CDs, DVDs, and Blu-ray discs for laser-read data storage.

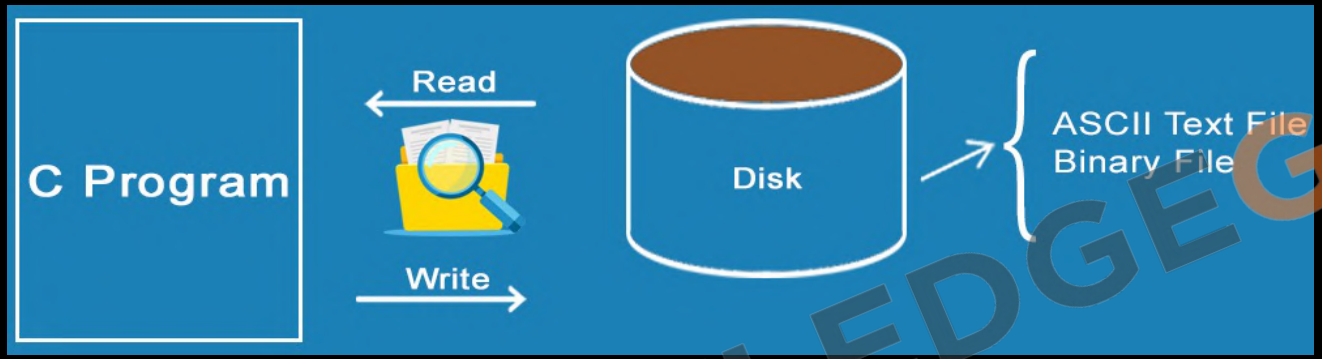


13.1 Data Organization



1. Simplicity: C library functions provide a simple interface for file operations, hiding complex OS details.
2. Portability: Programs use the same library calls regardless of the underlying OS, enhancing code portability.
3. Standardization: The C Standard Library ensures consistent behavior across different environments.
4. Ease of Use: Developers can perform file operations without knowing OS-specific system calls.
5. Abstraction: The library acts as an abstraction layer, managing OS interactions and disk I/O seamlessly.

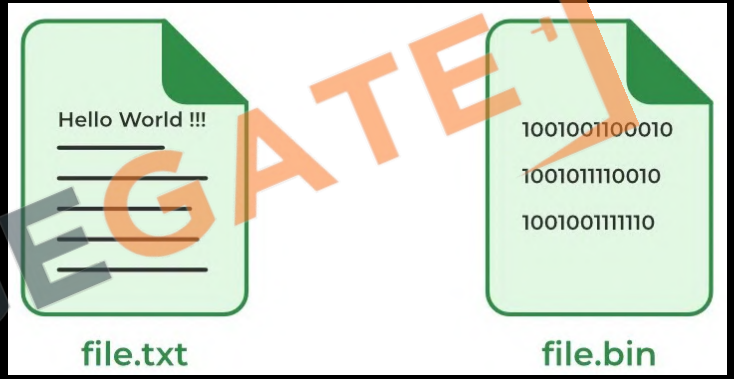
13.2 File Operations



1. Creation of a New File: Establishes a new file on disk, typically opened in write or append mode.
2. Opening an Existing File: Acquires access to an existing file.
3. Reading from a File: Retrieves data from a file, usually into a program's buffer.
4. Writing to a File: Outputs data to a file, storing it on disk from the program's buffer.
5. Moving to a Specific Location in a File (Seeking): Changes the current position of the file pointer to a specified location for subsequent operations.
6. Closing a File: Releases the file, ensuring all buffers are flushed and resources are freed.

13.3 Text Files & Binary Files

1. Encoding: Text files store data in human-readable format while binary files store data in the same format as the in-memory representation.
2. Data Interpretation: Text files represent data as characters, whereas binary files represent data as a sequence of bytes.
3. Portability: Text files are more portable across different platforms. Binary files can be platform-specific due to different data representations.
4. Efficiency: Binary files are generally more efficient for I/O operations because they don't require translation.



13.4 File pointer

```c
#include <stdio.h>

int main() {
  FILE *fp; // DECLARING A FILE POINTER
  char buf[1024];

  // OPENING FILE WITH FILE NAME AND MODE
  fp = fopen("kgcoding.txt", "r");

  // READING FROM FILE
  fgets(buf, sizeof(buf), fp);

  printf("File contents: %s\n", buf);

  // IMPORTANT! MUST CLOSE FILE POINTER
  fclose(fp);
  return 0;
}
```

1. Variable Type: A file pointer is of type FILE* and is used to reference a file.
2. Standard Files: C automatically provides file pointers like stdin, stdout, and stderr.
3. Opening Files: File pointers are associated with a file on disk through the fopen function.
4. Reading/Writing: They are used for reading from or writing to files with functions like fread, fwrite, fprintf, fscanf, etc.
5. Position Tracking: Keeps track of the current position within the file for read/write operations.
6. Closing Files: Should be passed to fclose to close the file and release resources.

13.5 Open a File

```c
#include <stdio.h>

int main() {
  FILE *fp; // DECLARING A FILE POINTER
  char buf[1024];

  // OPENING FILE WITH FILE NAME AND MODE
  fp = fopen("kgcoding.txt", "r");
  if (fp == NULL) {
    printf("Error opening file");
    return 1;
  }

  // READING FROM FILE
  fgets(buf, sizeof(buf), fp);
  printf("File contents: %s\n", buf);

  // IMPORTANT! MUST CLOSE FILE POINTER
  fclose(fp);
  return 0;
}
```

1. fopen() Function: Opens a file and returns a FILE* pointer for access.
2. Modes: Specify mode (r, w, a, etc.) to open for reading, writing, appending, etc.
3. Null on Failure: Returns NULL if the file cannot be opened.
4. Error Checking: Always check for NULL to confirm file was opened successfully.
5. Path: Can open files using relative or absolute paths.

13.6 Close a File

```c
#include <stdio.h>

int main() {
  FILE *fp; // DECLARING A FILE POINTER
  char buf[1024];

  // OPENING FILE WITH FILE NAME AND MODE
  fp = fopen("kgcoding.txt", "r");
  if (fp == NULL) {
    printf("Error opening file");
    return 1;
  }

  // READING FROM FILE
  fgets(buf, sizeof(buf), fp);
  printf("File contents: %s\n", buf);

  // IMPORTANT! MUST CLOSE FILE POINTER
  fclose(fp);
  return 0;
}
```

1. fclose() Function: Used to close an open file pointed to by a FILE* pointer.
2. Release Resources: Frees up the system resources associated with the file.
3. Flush Buffer: Writes any remaining data in the file buffer to the file.
4. Nullify Pointer: Good practice to set the file pointer to NULL after closing.
5. Error Handling: Check the return value for successful closure (0 is successful, EOF is an error).

13.7 Reading data from File

```c
#include <stdio.h>

int main() {
  FILE *fp; // DECLARING A FILE POINTER
  char buf[1024];

  // OPENING FILE WITH FILE NAME AND MODE
  fp = fopen("kgcoding.txt", "r");
  if (fp == NULL) {
    printf("Error opening file");
    return 1;
  }

  // READING FROM FILE
  fgets(buf, sizeof(buf), fp);
  printf("File contents: %s\n", buf);

  // IMPORTANT! MUST CLOSE FILE POINTER
  fclose(fp);
  return 0;
}
```

1. Functions: Use fgetc, fgets, fread, or fscanf to read from files.
2. Modes: Open the file in read mode ("r") or read/update mode ("r+").
3. Buffer Management: Ensure buffers are properly sized, especially with fgets and fread.
4. Binary vs. Text: Choose read functions based on whether the file is binary or text.

13.8 End of File (EOF)

```c
#include <stdio.h>

int main() {
    FILE *filePointer = fopen("kgcoding.txt", "r");
    if (filePointer == NULL) {
        perror("Error opening file");
        return 1;
    }

    int ch;
    // Continue until EOF
    while ((ch = fgetc(filePointer)) != EOF) {
        putchar(ch);
    }

    fclose(filePointer);
    return 0;
}
```

1. Indicator: EOF is a constant used to indicate the end of a file or an error.
2. Value: Typically defined as -1 in C libraries.
3. Stream Checking: Functions like fgetc() use EOF to signal end of input stream.
4. Error Signaling: Can also signal an error if a file operation fails.
5. Loop Control: Often used in loops reading from files to determine when to stop reading.

13.9 Writing data to a File

```c
#include <stdio.h>

int main() {
  // Open the file for writing
  FILE *filePointer = fopen("kgcoding.txt", "w");
  if (filePointer == NULL) {
      printf("Error opening file");
      return 1;
  }

  // Write formatted text to the file
  fprintf(filePointer, "KGCoding is the best!\n");
  fprintf(filePointer, "The square of %d is %d.\n", 5, 5*5);

  fclose(filePointer); // Close the file
  return 0;
}
```

1. Functions: Use fputc, fputs, fwrite, or fprintf for writing data.
2. Open Modes: Open file in write ("w"), append ("a"), or update ("w+", "a+") mode.
3. Buffer Flushing: fflush() forces write buffer to flush, ensuring data is saved.
4. Text vs. Binary: Choose write function and mode based on file type (text or binary).

13.10 Appending data to a File

```c
#include <stdio.h>

int main() {
    FILE *filePointer = fopen("kgcoding.txt", "a");
    if (filePointer == NULL) {
        perror("Error opening file");
        return 1;
    }

    // Append text to the file
    fprintf(filePointer, "Appending a new line.\n");
    fprintf(filePointer, "The sum of %d and %d is %d.\n", 3,
    7, 3+7);

    fclose(filePointer);
    return 0;
}
```

13.11 File Opening Modes

File Mode	Meaning of Mode	During Inexistence of file
r	Open for reading.	If the file does not exist, fopen() returns NULL.
w	Open for writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a	Open for append. i.e, Data is added to end of file.	If the file does not exists, it will be created.
r+	Open for both reading and writing.	If the file does not exist, fopen() returns NULL.
w+	Open for both reading and writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a+	Open for both reading and appending.	If the file does not exists, it will be created.

Revision

1. Data Organization
2. File Operations
3. Text Files & Binary Files
4. File pointer
5. Open a File
6. Close a File
7. Reading data from File
8. End of File (EOF)
9. Writing data to a File
10. Appending data to a File
11. File Opening Modes

CHALLENGE

99. Write a program that asks the user for a filename, attempts to open it, and reports whether the operation was successful or not.
100. Create a program that reads integers from a file and calculates their sum.
101. Write a program that copies one text file's contents to another, stopping when it reaches EOF of the source file.
102. Write a program that takes user input and writes it to a file, ensuring each entry is on a new line.
103. Develop a program that appends user input to the end of a log file each time it's run.
104. Create a program that performs both reading and writing operations on a file called data.txt

KG Coding

Some Other One shot Video Links:

- Complete Java
- Complete HTML & CSS
- Complete JavaScript
- Complete React and Redux
- One shot University Exam Series

http://www.kgcoding.in/

Our YouTube Channels

KG Coding Android App

KG Coding

Knowledge GATE

KG Placement Prep

Sanchit Socket