Green University of Bangladesh

Department of Computer Science and Engineering (CSE)

Faculty of Sciences and Engineering

Semester : (Fall, Year : 2023), B.Sc. in CSE (Day)

Assignment(Group-J)

Course Title : Structured Programming

Course Code : CSE 103   Section : 231-D1

Student Details

	Name	ID
1.	Md.Miadul Islam Nizzan	231902032
2	Jenifar Binte Mahbub	231902043
3.	kashpia kamal	231902017
4.	Sineya Samarum Adhora	231902030
5.	Moynul Islam Anik	231902020

Assigned Date : 26.12.23

Submission Date : 05.01.24

Course Teacher's Name : Prof. Dr. Md. Saiful Azad

Assignment Status

Marks: ………………………………… **Signature:.....................**

Comments:.. **Date:.............................**

Multiple Choice Questions-

1. 32
2. stdio.h
3. D. All of these
4. ||
5. A. (type)
6. stdio.h
7. A. getchar() and putchar()
8. D. Both (B) and (C)
9. A. 50 50 50
10. D. %f
11. A. r= 3
12. A. copies s1 string into s2
13. B. 50 40 30 20 10
14. D. function
15. C. NFO
16. B. //
17. C. I = 22
18. D. casting
19. A. Square = 144
20. C. void
21. A. x= 15
22. C. Both (A) and (B)
23. -1
24. D. complex
25. C. 12 12 11 11
26. A. (*a).b
27. A. Info world
28. B. strlen
29. 4 bytes
30. hcvt
31. B. 1
32. A. #nifdefn
33. A. 22
34. A. x is a pointer to pointer
35. A. 00
36. B. \0
37. B. 48 48
38. A. 0
39. C. Line 3
40. A. 10
41. A. realloc
42. B. 4 bytes
43. B. Good by

44. A. !
45. ello world
    ello world
46. None of these
47. a=300
48. C. goto
49. C. BCPL
50. A. Dennis Ritchie
51. C. 1972
52. C. Middle Level
53. D. All of these
54. B. #
55. A. Float
56. D. All of the above
57. B. -32768 to +32767
58. D. ;
59. D. \n
60. A. \a
61. A. Single quotes
62. B. Double quotes
63. A. ‘’
64. 32
65. D. 8 bytes
66. D. 16 bytes
67. B. 4 bytes
68. None
69. B. a += 5
70. C. Logical
71. A. Bitwise AND
72. D. All of these
73. D. ==
74. B. is used first
75. A. Masking
76. A. set the desired bits to 1
77. C. =⟩
78. A. Right to Left
79. &&
80.
81. C. evaluated first
82. B. truncating the fractional part
83. A. ?
84. 5
85. A. (type)
86. A. Casting

87. A. a = a + 4
88. B. p++ is a single instruction
89. A. C Library functions provide I/O facilities
90. C.
91. A) getchar() and putchar()
92. B) Zero
93. C) putchar("x")
94. A) getchar() and putchar()
95. D) Error message
96. C) Address
97. A) #define
98. B) \0
99. A) string.h
100. B) stdlib.h
101. B) stdio.h
102. B) 3
103. A) ;
104. B) 0
105. C) goto
106. A) to go to the next iteration in a loop
107. C) ?
108. A) string.h
109. D) None of the above
110. A) An unsigned decimal integer
111. D) Arithmetic, relational, assignment
112. B) external
113. C) a character
114. B) Six
115. B) curly braces
116. D) an analyzing tool in C
117. C) do
118. D) None of these
119. D) both a & b
120. A) array of pointers rather than as pointers to a group of contiguous array
121. A) programmer
122. C) both a and b
123. D) None of these
124. D) 40
125. A) Extra values are being ignored
126. C) 0

127. C) can be displayed as a single entity
128. A) Addition of float value to a pointer

Very Short Questions:

1. Variable: A named storage location that can hold values and change later in the program.
2. Constant: A named value that cannot be changed after it is defined.
3. Integer: 4 bytes
4. Float: 4 bytes
5. Char: 1 byte
6. Double: 8 bytes
7. Variable: Can change, constant: fixed.
8. Logical Variable: Holds one of two values, typically true or false.
9. Global Variable: Accessible throughout the program.
10. Word: Usually 4 bytes, depending on architecture.
11. Byte: 8 bits (smallest addressable unit).
12. Compiler errors, runtime errors, debugging tools, logical reasoning.
13. Binary sequence of instructions specific to the processor hardware.
14. No, needs compiled first into machine code.
15. Software that translates source code into machine code or another language.

16. To convert source code into an understandable form for the computer.
17. Difficult to read/write, error-prone, platform-specific, inflexible.
18. `condition ? then_expression : else_expression`
19. `<`, `>`, `<=`, `>=`, `==`, `!=`
20. `&&`, `||`, `!`
21. `&`, `|`, `^`, `~`, `<<`, `>>`
22. `getchar()`, `putchar()`, `gets()`, `puts()`
23. `printf()`, `scanf()`, `sprintf()`, `sscanf()`
24. Reads a single character from standard input.
25. Reads a single character from standard input without echoing it.
26. Reads a single character from standard input with echoing it.
27. Performs operations on files and disks.
28. Performs input/output on the console.
29. `if (condition)` followed by statements
30. `if (condition)` followed by statements, `else` followed by alternate statements.
31. Nested `if` statements inside another `if` block.
32. Set of instructions to perform a specific task.
33. Loop within a loop.
34. Loop that continues until a specific condition is met.
35. Error in the program syntax.
36. Error in the program logic that doesn't cause a crash.
37. Error that occurs during program execution.
38. A collection of elements of the same type stored sequentially in memory.
39. `type name[size];` (e.g., `int numbers[10];`)

40. `type name[rows][columns];` (e.g., `int table[3][4];`)
41. A reusable block of code with named parameters and return value.
42. Function included in the C library.
43. To return a value from a function to the calling code.
44. Concatenates two strings.
45. Compares two strings and returns their relative order.
46. Reverses the order of characters in a string.
47. Finds the length of a string.
48. Copies one string to another.
49. A function that calls itself.
50. Passing a value by copying it into the function, original remains unchanged.

51. Call by reference: Passing the address of a variable to a function, allowing the function to directly modify the original variable's value.

52. Pointer: A variable that stores the memory address of another variable.

53. Structure: A user-defined composite data type that groups variables of different types under a single name.

54. Structure vs. union:

- Structure: Members share a common name but have distinct memory locations.
- Union: All members share the same memory location, so only one member can have a value at a time.

55. typedef: Creates aliases for existing data types, improving readability and maintainability.

56. Structured programming and errors: Yes, it helps reduce errors by enforcing clear control flow and modularity.

57. Preprocessor: Processes code before compilation, performing tasks like text replacement, conditional compilation, and file inclusion.

58. Two features of the preprocessor:

- Macro definition (#define)
- File inclusion (#include)

59. Syntax for defining a file:

```
FILE *fileptr = fopen("filename", "mode");
```

60. Syntax for opening a file:

```
FILE *fileptr;
fileptr = fopen("filename", "mode");
```

Here are concise answers to your short questions:

61. Closing a file: `fclose(file_pointer);`

62. fopen(): Opens a file, returns a file pointer.

63. fclose(): Closes a file, releases resources.

64. Three constants: Integer, floating-point, character.

65. Bitwise left shift: Shifts bits left, multiplies by 2.

66. Unary operator: Operates on one operand (e.g., -x, !y).

67. putchar(): Outputs a single character.

68. Expression: A combination of values, variables, operators.

69. Expression vs. variable: Expressions yield a value, variables store a value.

70. Primary data types: int, float, char, double, void.

71. C as mid-level language: Combines high-level features with low-level control.

72. Escape sequence characters: Represent special characters (e.g., \n for newline).

73. if statement: Executes code conditionally.

74. while loop syntax: `while (condition) { code; }`

75. Output: 2 2 3 4

76. Loop types: for, while, do-while.

77. for loop: Executes code a specific number of times.

78. Format specifiers: %d, %f, %c, %s, etc.

79. Output: content of array 11 12 13 14 15 16 17

80. Declaring a 1D array: `data_type array_name[size];`

81. Multidimensional array: An array with multiple dimensions, arranged as a matrix or higher-order structure. For example, a 2D array represents rows and columns, like a table.

82. Function explanations:

- strlen(str): Returns the length of the string `str`, excluding the null terminator.
- strcat(dest, src): Appends the string `src` to the end of the string `dest`.

83. Character I/O functions:

- getc(fp): Reads a single character from the file pointed to by `fp`.
- putc(ch, fp): Writes the character `ch` to the file pointed to by `fp`.

84. File handling functions:

- fopen(filename, mode): Opens a file named `filename` in the specified `mode` (e.g., "r" for reading, "w" for writing).
- fclose(fp): Closes a file previously opened with `fopen()`.

85. Macros: Text substitution mechanisms defined using the `#define` directive. They replace a macro name with its defined value during the preprocessing stage.

86. #include: A preprocessor directive that includes the contents of a specified header file into the current file. Header files contain function declarations, macro definitions, and other shared code.

87. Dynamic memory allocation: The process of allocating memory at runtime using functions like `malloc()`, `calloc()`, and `realloc()`. This allows for flexible memory usage based on program needs.

88. Advantages of functions:

- Modularity: Break down complex problems into smaller, manageable functions.
- Reusability: Use the same function in multiple parts of the code or even in different programs.
- Abstraction: Hide implementation details and make code easier to understand and maintain.

89. Purpose of void keyword:

- Used to indicate no return value from a function.
- Used to specify a generic pointer type that can point to any data type.

90. Expression evaluation:

- a) Cannot be determined without values for d and c.
- b) True (1), as i is greater than 0 and j is less than 5.

91. Associativity: The order in which operators of the same precedence are evaluated in an expression. For example, multiplication and division have left-to-right associativity.

92. Expression evaluation:

- a) 21.85
- b) 5.057143

93. Arithmetic operators: Perform mathematical operations on numerical values (+, -, *, /, % for addition, subtraction, multiplication, division, and modulo).

94. scanf(): Reads formatted input from the standard input stream (stdin). For example, `scanf("%d %f", &

95.**Difference between formatted & unformatted statements?**

- **Formatted statements** involve output functions like **printf** where you specify the format of the output using format specifiers (**%d**, **%f**, **%s**, etc.). It gives you control over the presentation of the output.
- **Unformatted statements** are typically used with functions like **putchar**, **puts**, or **fprintf** without specific format specifiers. They output data without formatting.

96.Features of pre-processor:

- Macro substitution: Replace defined macros with their corresponding values.
- File inclusion: Include files into the source code using **#include**.
- Conditional compilation: Control compilation based on conditions with **#if**, **#ifdef**, **#ifndef**.
- Compiler directive control: Directives like **#pragma** can control compilation aspects.

97.Command line argument:

- It refers to the parameters passed to a program when it's run through the command line interface. These parameters can be used to modify the behavior of the program dynamically during execution.

98. **Variable & constant:**

- **Variable:** A storage location in the computer's memory where you can store and manipulate data. Its value can change during the execution of the program.
- **Constant:** A value that doesn't change during program execution. Constants can be of various types like numeric constants (e.g., **3.14**), character constants (e.g., **'a'**), or symbolic constants (e.g., **#define PI 3.14**).

99.How static variables are defined and initialized:

- Static variables in C are defined using the **static** keyword within a function.
- They are initialized only once before the program starts execution and retain their value between function calls.

100. Storage class of variables:

- **The storage class of a variable determines the scope, visibility, and lifetime of the variable within a C program. Common storage classes include auto, register, static, and extern.**
-

101.Three file modes in C:

- **Read ("r"): Opens a file for reading.**
- **Write ("w"): Opens a file for writing. If the file exists, it truncates the file to zero length. If the file doesn't exist, it creates a new file.**
- **Append ("a"): Opens a file for writing at the end of the file. If the file doesn't exist, it creates a new file.**

102.Program to find the largest between two numbers:

```
#include <stdio.h>
int main() {
    int n1, n2;
    printf("Enter two numbers: ");
    scanf("%d %d", &n1, &n2);

    if (n1 > n2) {
        printf("Largest number is: %d\n", n1);
    } else if (n2 > n1) {
        printf("Largest number is: %d\n", n2);
    } else {
        printf("Both numbers are equal\n");
    }

    return 0;
}
```

Short Questions:

1. Features of C:

- Structured programming: Encourages modularity and code readability.

▪Mid-level language: Balances low-level hardware control and high-level readability.

- Portable: Code can be compiled and run on different platforms with minimal changes.
- Rich set of operators: Supports various arithmetic, logical, relational, bitwise, and other operations.
- Efficient: Produces compact and fast code.
- Procedural: Focuses on step-by-step procedures using functions.
- General-purpose: Used for a wide range of applications, from system programming to application development.

2. Data types in C:

- Basic types: `int`, `float`, `double`, `char`, `void`
- Derived types: Arrays, pointers, structures, unions
- User-defined types: Enumerated types (`enum`), typedef

3. Type identifiers in C:

- Keywords used to declare variables of specific data types (e.g., `int`, `float`, `char`).

4. Structure of C program:

- Preprocessor directives: Instructions for the preprocessor (e.g., `#include`, `#define`)
- Main function: `int main()`, the program's entry point
- Declarations: Variables, functions, constants
- Statements: Instructions that perform actions
- Comments: Explanatory text ignored by the compiler

5. Operators in C:

- Arithmetic: +, -, *, /, %
- Relational: ==, !=, <, >, <=, >=
- Logical: && (AND), || (OR), ! (NOT)
- Bitwise: &, |, ^, ~, <<, >>
- Assignment: =
- Increment/decrement: ++, --
- Conditional: ?:
- Comma: ,

6. Four data types in C:

- int: Stores integers (whole numbers)
- float: Stores single-precision floating-point numbers (decimals)
- double: Stores double-precision floating-point numbers (higher precision decimals)
- char: Stores characters (single letters, symbols)

7. Difference between = and ==:

- = (assignment operator): Assigns a value to a variable (e.g., `x = 5`)
- == (equality operator): Compares two values for equality, returns true (1) if equal, false (0) otherwise (e.g., `if (x == 5)`).

8. Precedence and order of evaluation:

- Operators have different precedence levels, determining the order of evaluation in expressions.
- Operators with higher precedence are evaluated before those with lower precedence.
- The order of evaluation can be altered using parentheses to force a specific order.
- Common precedence (high to low):

    - Parentheses ()
    - Unary operators (e.g., !, ++, --)
    - Arithmetic (*, /, %, +, -)
    - Relational (==, !=, <, >, <=, >=)
    - Logical (&&, ||)
    - Assignment (=)

I'll continue answering the C programming short questions 10-20.

10. Input & Output functions:

- printf(): Outputs formatted data to the console. Example: `printf("Hello, %s!\n", name);`
- scanf(): Reads formatted input from the console. Example: `scanf("%d %f", &age, &height);`

11. Variables:

- Named storage locations holding values of specific data types.
- Rules for defining:
    - Choose a meaningful name (letters, digits, underscores).
    - Start with a letter or underscore.
    - Declare with data type and name (e.g., `int age;`).
    - Assign a value (optional).

12. Local vs. global variables:

- Local: Declared within a function, accessible only within that function.
- Global: Declared outside functions, accessible throughout the program.

13. Symbolic constants:

- Constants represented by names using `#define`. Example: `#define PI 3.14159`

14. Bitwise operators:

- & (bitwise AND): Performs AND operation on each bit of operands. Example: `a & b`
- | (bitwise OR): Performs OR operation on each bit. Example: `a | b`

15. ++i vs. i++:

- ++i (pre-increment): Increments i first, then uses the new value.
- i++ (post-increment): Uses the current value of i, then increments it.

16. Logical operators:

- && (AND): True if both operands are true.
- || (OR): True if either or both operands are true.
- ! (NOT): True if the operand is false.

17. Functions:

- getch(): Reads a single character from the console without echoing it.
- clrscr(): Clears the console screen (non-standard, might require specific libraries).

18. printf():

- Outputs formatted data to the console.
- Example: `printf("Age: %d, Height: %.2f\n", age, height);`

19. scanf():

- Reads formatted input from the console.
- Example: `scanf("%d %f", &age, &height);`

20. Do While statement:

- Syntax: `do { statements; } while (condition);`
- Executes the statements at least once, then repeats as long as the condition is true.

21. Looping statements in C:

- for loop: Executes a block of code a specific number of times. Syntax: `for (initialization; condition; increment/decrement) { statements; }` Example: `for (int i = 0; i < 10; i++) { printf("%d ", i); }`
- while loop: Repeats a block of code as long as a condition is true. Syntax: `while (condition) { statements; }` Example: `while (x > 0) { x = x / 2; }`
- do-while loop: Executes a block of code at least once, then repeats as long as a condition is true. Syntax: `do { statements; } while (condition);`

22. continue and break statements:

- continue: Skips the remaining statements in the current iteration of a loop and jumps to the next iteration.
- break: Exits a loop immediately, even if the condition is still true.

23. switch statement:

- Selects one of multiple code blocks to execute based on the value of an expression. Syntax: `switch (expression) { case value1: statements;`

`break; case value2: statements; break; ... default: statements; }`
Example: `switch (grade) { case 'A': printf("Excellent!\n"); break; case 'B': printf("Good job!\n"); break; ... }`

24. Nested if-else:

- An if or else statement within another if or else statement, creating multiple decision levels. Example: `if (x > 0) { if (y > 0) { printf("Both positive\n"); } else { printf("X positive, Y non-positive\n"); } }`

25. Nested for loop:

- A for loop within another for loop, creating nested iterations. Example: `for (int i = 0; i < 3; i++) { for (int j = 0; j < 2; j++) { printf("%d %d\n", i, j); } }`

26. Array:

- A collection of elements of the same data type, stored under a common name.
- Declaration: `data_type array_name[size];` Example: `int numbers[5];`

27. One-dimensional array:

- A linear array with elements accessed using a single index. Example: `int

I'll continue answering the C programming short questions 28-39.

28. Two-dimensional array:

- A matrix-like array with elements accessed using two indices (rows and columns).

- Example: `int matrix[3][4];`
- Accessing elements: `matrix[row][column]`

29. Applications of arrays:

- Storing lists of items (e.g., names, grades)
- Representing matrices and tables
- Implementing stacks, queues, and other data structures
- Managing large datasets
- Image and signal processing

30. String functions:

- strlen(str): Returns the length of a string (excluding null terminator).
- strcpy(dest, src): Copies a string from `src` to `dest`.
- strcat(dest, src): Appends `src` to the end of `dest`.
- strcmp(str1, str2): Compares two strings, returning 0 if equal.

31. Call by value vs. call by reference:

- Call by value: Copies the value of an argument to the function's parameter, changes within the function don't affect the original variable.
- Call by reference: Passes the memory address of the argument, allowing the function to modify the original variable.

32. Recursion:

- A function calling itself directly or indirectly, used for tasks that can be broken down into smaller, self-similar subproblems.
- Example: Fact calculation (recursive): `int fact(int n) { if (n == 0) { return 1; } else { return n * fact(n-1); } }`

33. Null terminator '\0' in strings:

- Marks the end of a string in C.
- Essential for string functions to work correctly.
- Example: `char name[] = "Alice\0";`

34. Function:

- A block of code that performs a specific task, reusable multiple times.
- Definition: `return_type function_name(parameters) { statements; }`

35. Calling function vs. called function:

- Calling function: The function that calls another function.
- Called function: The function being invoked by the calling function.

36. Void function:

- A function that doesn't return a value.
- Used for tasks that only perform actions, not calculations.

37. Pointer:

- A variable that stores the memory address of another variable.
- Declaration: `data_type *pointer_name;`
- Example: `int x = 10; int *ptr = &x;`

38. Pointer to structure:

- A pointer that points to a structure variable.
- Accessing members: `pointer->member_name`

39. Pointer to function:

- A pointer that points to a function's memory address.
- Used for function callbacks and dynamic function calls.

I40. Let's try again with a concise explanation of malloc(), a widely used dynamic memory allocation function in C:

malloc():

- Purpose: Allocates a block of memory of a specified size during program execution.
- How it works: Requests memory from the heap (a pool of free memory).
- Return value: A pointer to the beginning of the allocated block, or `NULL` if allocation fails.
- Key points:
  - Need to cast the returned pointer to the desired data type.
  - Always check for `NULL` to handle allocation failures.
  - Use `free(ptr)` to release the allocated memory when done.
-
- Benefits: Enables flexible data structures and adapts to varying memory needs during runtime.

Example:

```
int *ptr = malloc(10 * sz(int));

if (ptr == NULL) {

  } else {
```

```
  free(ptr);}
```

Hope this provides a clearer understanding of dynamic memory allocation using `malloc()` in C.

41. Accessing value using pointer:

- Use the `*` operator to dereference the pointer and access the value it points to.
- Example:

```
int x = 10;
int *ptr = &x;            // ptr points to x int value =
*ptr;                     // value will be 10
```

42. Pointer to pointer:

- A pointer that stores the address of another pointer.
- Declaration: `data_type ptr_to_ptr;`
- Used for multi-level referencing and dynamic memory allocation.

43. malloc vs. calloc:

- Both allocate memory dynamically.
- malloc: Only allocates memory, doesn't initialize it.
- calloc: Allocates and initializes all bytes to zero.

44. Structure:

- A user-defined composite data type that groups variables of different types under a single name.
- Example:

```
struct Student {
	int id;
	char name[50];
	float gpa;
};
```

45. Array of structures:

- An array where each element is a structure.
- Example:

```
struct Student students[10];
```

46. Nested structure:

- A structure containing another structure as a member.
- Example:

```
struct Address {
	char city[50];
	char state[2];
	int zip;
};

struct Employee {
	int id;
	char name[50];
	struct Address address;
};
```

47. Structure vs. union:

- Structure: All members have their own memory allocation.
- Union: All members share the same memory location, only one member can have a value at a time.

48. Structure vs. array:

- Structure: Groups variables of different types, accessed using member names.
- Array: Stores elements of the same type, accessed using indices.

49. Programming approach:

- A systematic way to solve problems using programming:
    1. Problem definition
    2. Algorithm design
    3. Coding
    4. Testing and debugging
    5. Documentation
-

50. typedef:

- Creates an alias for an existing data type, making code more readable and maintainable.
- Example:

```
typedef unsigned int UINT;
typedef struct Student Person;
```

Here are the answers to your questions:

51. Explain directives.

- Directives are instructions given to the preprocessor, a program that processes code before the actual compilation stage.
- They are indicated by a pound sign (#) at the beginning of a line.

- They don't directly produce executable code, but they modify the source code to influence the compilation process.
- Common directives include #include, #define, #ifdef, #ifndef, #endif, etc.

52. Explain features of preprocessors.

- Text Substitution: Replaces text patterns with other text, commonly used for defining macros and constants.
- File Inclusion: Inserts the contents of external files (header files) into the current file, promoting code reusability and modularity.
- Conditional Compilation: Controls which parts of the code are compiled based on conditions, enabling code customization and debugging options.
- Macro Expansion: Defines macros (text shortcuts) that can be expanded inline, simplifying code and reducing redundancy.

53. What is a pre-processor? Explain #include and #define.

- Pre-processor: A program that modifies source code before compilation, often used for tasks like file inclusion, macro definition, conditional compilation, and more.
- #include: Inserts the contents of a specified header file into the current file, making its declarations and definitions available.
- Example: `#include <stdio.h>` includes the standard input/output header file.
- #define: Defines a macro, replacing a text pattern with another text throughout the code.
- Example: `#define PI 3.14159` defines a macro named "PI" with the value 3.14159.

54. C Preprocessors

- Definition: A preprocessor is a directive that instructs the compiler to perform specific tasks before the actual compilation of code begins. It's a text-substitution tool that processes code before the compiler sees it.
- Purpose:
    - Include header files (#include)
    - Define constants (#define)
    - Conditional compilation (#ifdef, #ifndef, #else, #endif)
    - Macros (#define)
    - Remove comments (/#)
-

55. File Handling in C

- Definition: File handling involves working with files to perform operations like reading, writing, and modifying data.
- Steps:
    1. Open the file using `fopen()`
    2. Perform operations (read/write) using functions like `fscanf()`, `fprintf()`, `fgetc()`, `fputc()`
    3. Close the file using `fclose()`
-

56. getch vs. getc

- getch():
    - Reads a single character from the keyboard without echoing it to the screen.
    - Often used to hold output for viewing until a key is pressed.
-
- getc():
    - Reads a single character from a specified file stream.

    - Can also be used to read from the keyboard by specifying stdin as the file stream.
-

57. putch vs. putc

- putch():
    - Writes a single character to the console without buffering.
    - May not be available in all compilers.
-
- putc():
    - Writes a single character to a specified file stream.
    - Can also be used to write to the console by specifying stdout as the file stream.
-

58. putc and getc in Brief

- putc(character, file_pointer): Writes a character to a file.
- getc(file_pointer): Reads a character from a file.

59. Command Line Arguments

- Definition: Additional information passed to a program when it's executed from the command line.
- Access: Using argc (argument count) and argv (argument vector) in the
- main() function.

60. Command Line Argument Example

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

```
int main(int argc, char *argv[]) {
    if (argc == 2) {
        printf("Hello, %s!\n", argv[1]);
    } else {
        printf("Please provide a name as a command line argument.\n");
    }
    return 0;
}
```

To run this program with a name as an argument:

```
./program_name John
```

Output:

```
Hello, John!
```

Here are the answers to your C programming questions:

61. Macros

- Definition: A macro is a preprocessor directive that defines a text substitution to be replaced in code before compilation.
- Example:

```
#define PI 3.14159

int area = PI * radius * radius;
```

-
- The preprocessor replaces all occurrences of `PI` with `3.14159` before compilation.

62. if-else-if vs. switch statement

- if-else-if: Used for multiple conditional checks, executing code based on the first true condition.
- switch statement: Used for multiple choices based on a single expression's value, jumping to a specific case label.
- Best choice:
  - Use `if-else-if` for complex conditions or non-integer expressions.
  - Use `switch` for simple conditions with integer or enumerated values.
-

63. Functions with arguments and return types

- Arguments: Values passed to a function for its operations.
- Return type: The data type of the value the function returns.
- Example:

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

64. Array of pointers

- Definition: An array where each element is a pointer to a variable of a specific data type.
- Example:
- C

```
int *numbers[5];
```

-

65. Arrays and two-dimensional array initialization

- Array: A collection of elements of the same data type, accessed using an index.
- Two-dimensional array initialization:

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

66. Dynamic memory allocation

- Definition: Allocating memory at runtime using `malloc()`, `calloc()`, or `realloc()`.
- Purpose: To create data structures with flexible size during program execution.

67. Pointer to pointer

- Definition: A pointer that stores the address of another pointer.
- Example:
- C

```
 int ptr = &num_ptr;
```

-

68. Structures and unions

- Similarities: Both group variables of different data types under a single name.
- Difference:
  - Structure: Each member has its own memory location.

    - Union: All members share the same memory location, with only one member active at a time.

-

69. Nested structures

- Definition: A structure containing another structure as a member.
- Example:

```
struct Address {
	char city[50];
	char state[30];
};

struct Person {
	char name[50];
	int age;
	struct Address address;
};
```

70. Array of structures

- Definition: An array where each element is a structure.
- Example:

struct Book books[10];

71. Pointer to Structure:

- A pointer to a structure stores the memory address of the structure's first member.

- It allows you to directly access and modify structure members using the arrow operator (->).

- Example:

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

struct Student *ptr;
  ptr = &student1;
printf("%s is %d years old.", ptr->name, ptr->age);
```

72. sz Operator:

- Returns the size of a variable or data type in bytes.
- Useful for memory allocation and ensuring compatibility between different systems.
- Example:

```
int num = 10;
printf("Size of int: %d bytes\n", sz(int)); printf("Size of num:
%d bytes\n", sz(num));
```

73. Conditional Operator:
●A shorthand for if-else statements.

●Syntax: condition ? expression1 : expression2
●Evaluates to expression1 if condition is true, otherwise evaluates to expression2.

Example:

int x = 10;
```
int y = (x > 5) ? 20 : 30;
```
74. User-defined Functions vs. Built-in Functions:

- User-defined functions: Created by the programmer to perform specific tasks.
- Built-in functions: Predefined functions provided by the C library (e.g., printf, scanf, strlen).

75. Null String:

- A string with a null character (`\0`) as its first character.
- It's considered empty and has a length of 0.

76. Union:

- A data type that allows storing different data types in the same memory location, but only one member can have a value at a time.
- Used to conserve memory or represent different types of data efficiently.
- Example:

```
union Data {
        int i;
        float f;
        char str[20];
};
```

77. Expressions in C:

- Combinations of variables, operators, and function calls that produce a value.
- Used in assignments, conditional statements, loops, and function calls.

78. Static Variables:

- Retain their value between function calls, unlike standard local variables.
- Declared with the keyword within a function.
- Example:

```
void func() {
        static int count = 0;
        count++;
        printf("Count: %d\n", count);
}
```

79. Rule for declaring character constant:

- Enclose a single character within single quotes.
- Examples: 'A', 'z', '5', '@'

80. Rule for declaring string constant:

- Enclose a sequence of characters within double quotes.
- Examples: "Hello, world!", "Bangladesh", "12345"

81. Rule for declaring numeric constant:

- Represent integer values without quotes.
- Examples: 10, -5, 0
-
- Represent decimal values with a decimal point.
- Examples: 3.14159, -2.5
-
- Use scientific notation for very large or small numbers.
- Examples: 1.23e5, -6.78e-3

82. Structure:

- A user-defined data type that groups variables of different types under a single name.
- Example:

```
struct student {
        char name[50];
        int age;
        float gpa;
};
```

83. * operator (dereference operator):

- Accesses the value stored at the memory address pointed to by a pointer variable.
- Example:

```
int x = 10;
int *ptr = &x; // ptr points to the address of x
printf("%d", *ptr); // Prints 10, the value at the address pointed
to by ptr
```

& operator (address-of operator):

- Returns the memory address of a variable.
- Example (see above)

84. Rule of period(.) operator:

- Used to access members of a structure or union.
- Example:

```
struct student s = {"John", 20, 3.85};
printf("%s is %d years old.", s.name, s.age);
```

85. EOF and BOF:

- EOF (End-of-File): A special marker indicating the end of a file.
- BOF (Beginning-of-File): A special marker indicating the beginning of a file.

86. EOF value:

- Usually represented by the integer value -1.

87. Identifiers and keywords:

- Identifiers: User-defined names for variables, functions, structures, etc. (age, sum, calculateArea).
- Keywords: Reserved words with specific meanings in the programming language (int, float, if, while).

88. Type casting:

- Converting a value from one data type to another.
- Example:

```
int x = 10;
float y = (float)x / 3;
```

89. Spping:

- Exchanging the values of two variables.
- Example:

```
int a = 5, b = 10;
int temp = a;

a = b;
b = temp;
```

90. Ternary operator and cast operator:

- Ternary operator: condition ? expression1 : expression2 (a shorthandfor if-else statements).

- Cast operator: (type_name)expression (explicitly converts a value to a specific data type).

91. String Constant vs. Character Constant:

String Constant:

- Enclosed in double quotes (" ").
- Represents a sequence of characters.
- Internally stored as an array of characters terminated by a null character ('\0').
- Example: "Hello, world!"

Character Constant:

- Enclosed in single quotes (' ').
- Represents a single character.
- Example: 'A', 'z', '5', '!'

92. Character Constant vs. Integer Constant:

Character Constant:

- Represents a single character.
- Internally stored as an ASCII code.
- Examples: 'A' (ASCII code 65), '9' (ASCII code 57)

Integer Constant:

- Represents a whole number.
- Examples: 10, -5, 0

93. Arithmetic Operators in C:

- Addition (+)
- Subtraction (-)
- Multiplication (*)
- Division (/)
- Modulus (%)

94. Associativity Rules:

- Determines the order in which operators of the same precedence are evaluated.
- C operators have either left-to-right or right-to-left associativity.
- Examples:
    - Multiplication and division have left-to-right associativity: 10 / 2 * 4 is evaluated as (10 / 2) * 4 = 20.
    - Assignment operators have right-to-left associativity: x = y = 5 assigns 5 to both x and y.
-

95. Comparison and Logical Operators:

- Comparison Operators: Compare values and produce a Boolean result (true or false).
    - Examples: == (equal to), != (not equal to), < (less than), > (greater than), <= (less than or equal to), >= (greater than or equal to)
-

- Logical Operators: Combine Boolean expressions to produce a new Boolean result.
    - Examples: && (AND), || (OR), ! (NOT)
-

96. Differences from Arithmetic and Assignment Operators:

- Arithmetic operators perform mathematical computations on numbers.
- Assignment operators assign a value to a variable.
- Comparison and logical operators evaluate conditions and relationships between values.

97. Operators for Comparison and Logical Decision Making:

- Comparison Operators: ==, !=, <, >, <=, >=
- Logical Operators: &&, ||, !

98. Equality Operator (==) vs. Assignment Operator (=):

- Equality Operator: Checks if two values are equal.
- Assignment Operator: Assigns a value to a variable.

99. Bitwise Operators:

- Bitwise AND (&): Performs a bitwise AND operation on each pair of corresponding bits in two operands.
- Bitwise OR (|): Performs a bitwise OR operation on each pair of corresponding bits.
- Bitwise XOR (^): Performs a bitwise XOR operation on each pair of corresponding bits.
- Bitwise Left Shift (<<): Shifts the bits of a value to the left by a specified number of positions.

- Bitwise Right Shift (>>): Shifts the bits of a value to the right by a specified number of positions.

100. Unary Operators:

- Operate on a single operand.
- Examples:
    - Increment (++x, x++)
    - Decrement (--x, x--)
    - Negation (-x)
    - Logical NOT (!x)
    - Address-of (&x)
    - Dereference (*x)
    - Sz (sz(x))
-

101. Distinguish between binary minus and unary minus.

- Binary minus (-): It's a subtraction operator that operates on two operands, producing the difference between them. Example: result = 10 - 5;
-
- Unary minus (-): It's a negation operator that negates the value of a single operand, flipping its sign. Example: result = -5;

102. What is modulus operator and how does it operate in C?

- Modulus operator (%): It gives the rm of a division operation. It workswith both integers and floating-point numbers. Example: rm = 17 % 4;

  103. What is an expression? How is an expression different from the variables?

- Expression: A combination of variables, operators, and function calls that evaluates to a value. Examples: x + y, 2 * (a - b)

- Variable: A named storage location holding a value. Examples: int age = 25;, char initial = 'A';

104. What are the different types of statements used in C?

- Expression statements: Evaluate an expression and discard the result.Example: x + y;

- Compound statements: Groups of statements enclosed in curly braces.Example: { int x = 10; printf("%d", x);
- Control flow statements: Alter the execution flow of a program x{ int x = 10; printf("%d", x); }

- Jump statements: Transfer control unconditionally (goto, break,

  continue, return)

105. What are the salient features of standard input and output files?

- Standard input (stdin): Default source of input, usually the keyboard.
- Standard output (stdout): Default destination for output, usually the console.
- Features:
  - Pre-opened and ready to use.
  - No need for explicit opening or closing.
  - Associated with file descriptors 0 (stdin) and 1 (stdout).
-

106. Explain the following statements: i) getchar() ii) putchar() iii) EOF

- getchar(): Reads a single character from stdin and returns its ASCII value.
- putchar(): Prints a single character to stdout.
- EOF: End-of-File marker, indicating no more input available.

107. What is the scanf() and how does it differ from the getchar()?

- scanf(): Reads formatted input from stdin according to specified format specifiers.
- getchar(): Reads only a single character.

108. What are the format codes used along with the scanf()? Display the various data types in C.

- Format codes: Specifiers like `%d`, `%f`, `%c`, etc., indicating expected data types.
- Data types in C: `int`, `float`, `char`, `double`, `void`, `short`, `long`, `signed`, `unsigned`, etc.

109. What is the printf() and compare with putchar().

- printf(): Prints formatted output to stdout.
- putchar(): Prints only a single character.

110. What is meant by conditional expression?

- An expression that evaluates to one of two values based on a condition. Often used in `if` statements and ternary operators.

111. What is looping in C? What are the advantages of looping?

- Looping: Repeating a block of code until a certain condition is met.
- Advantages:
  - Automating repetitive tasks.
  - Processing large amounts of data efficiently.
  - Implementing algorithms that require iteration.

112. Nested for loop

- It's a loop within another loop, allowing for more complex iterations.
- The inner loop executes completely for each iteration of the outer loop.
- Example:

```
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 2; j++) {
    printf("i = %d, j = %d\n", i, j);
    }
}
```

113. While loop vs. for loop

- While loop: Repeats a block of code as long as a condition is true.
  - Syntax: while (condition) { code block }
  - Example: while (count < 10) { count++; }
- For loop: Repeats a block of code a specific number of times.
  - Syntax: for (initialization; condition; increment/decrement) { code block }
  - Example: for (int i = 0; i < 5; i++) { printf("%d ", i); }

114. Importance of main() in C

- It's the entry point of every C program.
- Execution begins from the main() function.
- Without it, the program won't run.

115. Use of continue in C

- Skips the remaining statements in the current iteration of a loop and jumps to the next iteration.
- Doesn't terminate the loop entirely.

116. Applications of C language

- System programming (operating systems, device drivers)
- Embedded systems (microcontrollers, appliances)
- Game development
- Computer graphics
- Data science and numerical computing
- Network programming
- Database management systems
- And many more

117. Advantages of functions

- Reusability of code
- Modularity (breaking down complex problems into smaller, manageable functions)
- Improved readability and maintainability
- Easier testing and debugging

118. Call by reference vs. call by value

- Call by reference: Passes the memory address of the argument to the function, allowing the function to modify the original value.
- Call by value: Passes a copy of the argument's value to the function, so any changes made within the function don't affect the original value.

119. Difference between call by reference and call by value

- Call by reference: Original value can be modified by the function.
- Call by value: Original value remains unchanged.

120. Purpose of return statement:

- It returns a value from a function to the calling code.
- It terminates the function's execution immediately.

121. Rules for return statement:

- Can only be used within a function.
- At most one return statement per function execution path.
- Must return a value matching the function's declared return type.
- If no return statement, control returns implicitly at the end.

122. Static variable:

- Retains its value between function calls.
- Declared with `static` keyword.
- Has scope within the function.

123. External data type:

- Declares a variable or function as accessible across multiple files.
- Declared with `extern` keyword.
- Definition must exist in one of the files.

124. Recursive function:

- Calls itself within its body.
- Merits: Elegant for problems with self-similar structure (e.g., tree traversals).
- Demerits: Can lead to stack overflow if not carefully designed.

125. fopen() function:

- Opens a file for reading or writing.
- Returns a file pointer if successful, NULL otherwise.
- Example: FILE *fp = fopen("myfile.txt", "r"); (opens for reading)

126. Array:

- Collection of elements of the same data type, stored contiguously in memory.
- Accessed using indices starting from 0.
- Array variable differs by holding multiple values, unlike a single value for ordinary variables.

127. Array indexing:

- Accessing individual elements using their position (index) within the array.
- Example: int arr[5] = {1, 2, 3, 4, 5}; int firstElement = arr[0].

128. Sing arrays:

- Not always necessary to use another array for sed elements.
- In-place sing algorithms modify the original array, saving memory.

129. Function:

- Reusable block of code that performs a specific task.
- Advantages: Modularity, code reusability, improved readability.
- Disadvantages: Overhead of function calls, potential scoping issues.

130. Function argument, call, and return value:

- Argument: Value passed to a function when it's called.

- Function call: Invoking a function's execution.
- Return value: Value sent back from a function to the calling code.

131. Automatic Variables

- Definition: Variables declared within a function or block, with local scope and automatic storage duration.
- Use:
  - Store temporary values within a function or block.
  - Preserve local variables across function calls (using recursion).
-

132. Initializing Automatic Variables

- Methods:
  - Direct assignment during declaration (int x = 10;).
  - Using initialization lists for arrays(int arr[] = {1, 2, 3};).
-

133. Initializing Static Variables

- Default Initialization: Zero for numeric types, null for pointers, empty string for character arrays.
- Explicit Initialization (Optional): During declaration or outside any function

134. External Data Type

- Purpose: Declare variables with global scope and file scope, accessible from multiple files.

- Declaration: Using the extern keyword (e.g., extern int global_var;).

135. Include Directive (#include)

- Purpose: Incorporate header files containing function prototypes, macros, and declarations.
- Usage: #include <header_name> for standard headers, #include
- "header_name" for user-defined headers.

136. Continuing #define Directive

- Methods:
  - Backslash(\) at the end of the line.
  - Enclosing the entire definition within parentheses.
-

137-139. Array Declaration Rules

- One-Dimensional: data_type array_name[size];
- Two-Dimensional: data_type array_name[rows][columns];
- Multi-Dimensional: data_type array_name[size1][size2]...[sizeN];

140. Character Array vs. Other Data Types

- Stores a sequence of characters, terminated by a null character (`\0`).
- Differs: Other data types store single values.

141. Character Array vs. String

- Character Array: A fundamental data structure for storing characters.
- String: A sequence of characters used to represent text, often implemented using character arrays.

142. Applications of Arrays

- Storing collections of data (e.g., lists, tables).
- Implementing data structures (e.g., stacks, queues).
- Representing matrices and multi-dimensional data.
- Text processing and string manipulation.

143. Pointers in C

- Variables that store memory addresses.
- Use:
  - Direct memory access and manipulation.
  - Dynamic memory allocation (e.g., using `malloc`).
  - Passing arguments by reference to functions.
-

144. Break Statement in Switch Statement

- Terminates the current `switch` block and jumps to the statement following it.
- Prevents fall-through to subsequent cases.

145. Array of Pointers vs. Pointer to Array

- Array of Pointers: An array where each element is a pointer to a different variable.
- Pointer to Array: A pointer that points to the first element of an array.

146. Purpose of string.h function:

- Houses a collection of functions for string manipulation in C programming.
- Key functions include:
  - `strcpy()`: Copies one string to another.

    - `strcat()`: Concatenates two strings.
    - `strlen()`: Determines string length.
    - `strcmp()`: Compares two strings.
    - `strstr()`: Finds a substring within a string.
-

147. Structure and its uses:

- A user-defined composite data type that groups variables of different types under a single name.
- Key uses include:
    - Representing complex data entities (e.g., employee records with name, age, salary).
    - Organizing code for better readability and maintainability.
-

148. Structure vs. other data types:

- Structure: Groups variables of different types under a single name.
- Other data types: Hold single values of a specific type (e.g., int, float, char).

149. Structure vs. array:

- Structure: Stores different types of data elements.
- Array: Stores a collection of elements of the same type.

150. Members or fields of structure:

- Individual variables within a structure, each with its own name and data type.
- Accessed using dot notation (e.g., struct_name.member_name).

151. Structure declaration vs. initialization:

Declaration:

- Defines the structure's template (name, members, data types).
- Doesn't allocate memory for variables.
- Example: `struct student { int id; char name[50]; };`

Initialization:

- Assigns values to members of a structure variable.
- Can be done during variable declaration or later.
- Example: `struct student s1 = {101, "John Doe"};`

152. Advantage of UNION in C:

- Allows storing different data types in the same memory location.
- Saves memory when only one member needs to be active at a time.
- Example: `union Data { int i; float f; char str[20]; };`

153. Salient features of typedef:

- Creates aliases for existing data types, improving readability and maintainability.
- Can be used with structures, unions, arrays, pointers, etc.
- Example: `typedef unsigned int UINT;`

154. Modes used in file operation:

- `r`: Open for reading (file must exist).
- `w`: Open for writing (creates new file or overwrites existing).
- `a`: Open for appending (adds data to the end).
- `r+`: Open for both reading and writing.
- `w+`: Open for reading and writing (overwrites existing).

- `a+`: Open for reading and appending.

155. Nested comments:

- Comments cannot be nested in C.
- Attempting to do so will cause a compilation error.

156. Binary vs. unary minus:

Binary minus:

- Subtracts two operands.
- Example: `a - b`

Unary minus:

- Negates a single operand.
- Example: `-x`

157. Modulus operator (%):

- Gives the rm of a division operation.
- Example: `10 % 3 = 1`

158. Why "goto" is not necessary in structured programming:

- Structured programming emphasizes control flow constructs like loops and conditionals.
- "goto" can lead to unstructured, hard-to-follow code.

159. Purpose of comma operator:

- Evaluates expressions from left to right, returning the value of the rightmost expression.
- Often used within for loops and macros.
- Example: for (i = 0, j = 10; i < j; i++, j--)
-

SAGMENT:

1. Write a C program to find the mximum of three numbers using conditional operators.

```
#include <stdio.h>

int main() {
    int n1, n2, n3, mx;

    printf("Enter the numbers: ");
    scanf("%d %d %d", &n1, &n2, &n3);


    if (n1 > n2) {
        if (n1 > n3) {
            mx = n1;
        } else {
            mx = n3;
        }
    } else {
        if (n2 > n3) {
            mx = n2;
        } else {
            mx = n3;
        }
    }
    printf("The mximum number : %d\n", mx);

    return 0;
}
```

2. Write a C Program to s an array in ascending Order.

```
#include <stdio.h>
void s-array(int arr[ ], int n) {

for (int i = 0; i < n - 1; i++) {
          for (int j = 0; j < n - i - 1; j++) {
                    if (arr[j] > arr[j + 1]) {
                    int temp = arr[j]; arr[j] = arr[j + 1]; arr[j + 1] = temp;
                  }
          }
}
int main() { int n;
printf("Enter the number of elements: ");

scanf("%d", &n);
int arr[n];

printf("Enter the elements of the array:\n");
 for (int i = 0; i < n; i++) {
scanf("%d", &arr[i]);
}
s_array(arr, n);
printf("Sed array in ascending order:\n");
for (int i = 0; i < n; i++) {
printf("%d ", arr[i]);
}
printf("\n");
return 0;
}
```

3. Write a C Program to s an array in descending Order

```
#include <stdio.h>

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int size = sz(arr) / sz(arr[0]);
    int temp;
    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - i - 1; j++) {
            if (arr[j] < arr[j + 1]) {
                temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }

    printf("Sorted array in descending order: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

4. Write a C Program to find sum of digits in a given number

```
#include <stdio.h>

int main() {
    int number, sum = 0;

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

    while (number > 0) {
        sum += number % 10;
        number /= 10
    }

    printf("Sum of digits: %d\n", sum);

    return 0;
}
```

5. Write a C Program to print square of all numbers
1 to 20 and print sum squares

```
#include <stdio.h>

int main() {
    int numbers[] = {3, 7, 12, 5, 9, 15};
    int nf = 5;
    int found = 0;
    for (int i = 0; i < sz(numbers) / sz(numbers[0]); i++) {
        if (numbers[i] == nf) {
            found = 1;
            break;
    }

    if (found) {
        printf("%d is present in the array.\n", nf);
    } else {
        printf("%d is not present in the array.\n", nf);
    }

    return 0;
}
```

7. Write a C Program to find the position of given
number in array

```
#include <stdio.h>

int main() {
    int numbers[] = {3, 7, 12, 5, 9, 15};
    int nf = 5;
    int p= 0;

    for (int i = 0; i < sz(numbers) / sz(numbers[0]); i++) {
        if (numbers[i] == nf) {
            printf("%d is present at position %d in the array.\n", nf, i);
            p = 1; // Set flag indicating the number is found
            break; // Exit loop since number is found
        }
    }

    if (!found) {
        printf("%d is not present in the array.\n", nf);
    }

    return 0;
}
```

8.Write a C Program to print transpose of matrix.

```
#include <stdio.h>
int main() {
    int r, c;
printf("Enter the number of rows and columns: ");
    scanf("%d %d", &r, &c);

int matrix[m][n];
printf("Enter the elements of the matrix:\n");
    for (int i = 0; i < r; i++) {
        for (j = 0; j < c; j++) {
            scanf("%d", &matrix[i][j]);
        }
    }
printf("Transpose of the matrix:\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            printf("%d ", matrix[j][i]);
        }
        printf("\n");
    }
 return 0;
}
```

9.Write a C Program to print equivalent binary number of given decimal number

```
#include <stdio.h>
int main() {
   int decimal, rm, binary = 0, i = 1;
printf("Enter a decimal number: ");
   scanf("%d", &decimal);
while (decimal != 0) {
       rm = decimal % 2;
       binary += rm * i;
       decimal /= 2;
       i *= 10;  // Shift the binary digits to the left
   }
  printf("Equivalent binary number: %d\n", binary);
 return 0;
}
```

10.Write a C Program to print equivalent octal number of given decimal number

```
#include <stdio.h>
int main() {
   int decimal, rm, binary = 0, i = 1;
printf("Enter a decimal number: ");
   scanf("%d", &decimal);
while (decimal != 0) {
       rm = decimal % 2;
       binary += rm * i;
       decimal /= 2;
       i *= 10;  // Shift the binary digits to the left
   }
printf("Equivalent binary number: %d\n", binary);
 return 0;
}
```

11.Write a C Program to print equivalent hex number of given decimal number

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

```
int main() {
    int decimal, rm;
    char hex[100];
    int i = 0;
    scanf("%d", &decimal);
while (decimal != 0) {
        rm = decimal % 16;
        if (rm < 10) {
 hex[i] = rm + 48;  // 48 is the ASCII code for '0'
        } else {
            hex[i] = rm + 55;  // 55 is the ASCII code for 'A'
        }
i++;
        decimal /= 16;
}

    hex[i] = '\0';  // Add null terminator
    for (int j = i - 1; j >= 0; j--) {
        printf("%c", hex[j]);
    }
printf("\n");
return 0;
}
```

12.Write a C Program to calculate fact of a given number using recursion

```
#include <stdio.h>
long fact(int n) {
   if (n == 0) {
       return 1;  // Base case: fact of 0 is 1
   } else {
       return n * fact(n - 1);  // Recursive call
   }
}
int main() {
   int num;
printf("Enter a non-negative number: ");
   scanf("%d", &num);
 if (num < 0) {
       printf("Fact is not defined for negative numbers.\n");
   } else {
       long result = fact(num);
       printf("Fact of %d = %ld\n", num, result);
   }
return 0;
}
```

13.Write a C Program to draw following object

No ans.because of no specific object.

14.Write a C Program to print all numbers between 1 to n divisible by 7

```
#include <stdio.h>
int main() {
    int n, i;
scanf("%d", &n);
```

```
    for (i = 1; i <= n; i++) {
        if (i % 7 == 0) {
            printf("%d ", i);
        }
    }
printf("\n");
    return 0;
}
```
15.Write a C Program to find sum of 1 + 2 + 3 +. . . .. + n
```
#include <stdio.h>
int main() {
    int n, i, sum = 0;
printf("Enter a positive integer: ");
    scanf("%d", &n);
for (i = 1; i <= n; i++) {
        sum += i;
    }
printf("Sum of 1 + 2 + ... + %d = %d\n", n, sum);
    return 0;
}
```
16.Write a C Program to find sum of 2 + 4 + 6 +. . . .. + n
```
#include <stdio.h>
int main() {
    int n, i, sum = 0;
 printf("Enter the value of n: ");
    scanf("%d", &n);
for (i = 2; i <= n; i += 2) {  // Iterate only over even numbers
        sum += i;
    }
 printf("Sum of 2 + 4 + 6 + ... + %d = %d\n", n, sum);
    return 0;
}
```
17.Write a C Program to find sum of 7 + 14 + 21 + ... + n
```
#include <stdio.h>
int main() {
    int n, i, sum = 0;
 printf("Enter the value of n (multiple of 7): ");
    scanf("%d", &n);
for (i = 7; i <= n; i += 7) {  // Iterate over multiples of 7
        sum += i;
    }
 printf("Sum of 7 + 14 + 21 + ... + %d = %d\n", n, sum);
    return 0;
}
```
18.Write a C Program to find sum of 1/1 + 1/2 + 1/3 + ... + 1/n
```
#include <stdio.h>
int main() {
    int n, i;
    float sum = 0.0;
printf("Enter the value of n: ");
    scanf("%d", &n);
for (i = 1; i <= n; i++) {
        sum += 1.0 / i;
}
 printf("Sum of harmonic series up to %d terms = %.4f\n", n, sum);
    return 0;
}
```

19.Write a C Program to print 15 terms of 1 , 2 , 4, 7, 11, 16, ...

```
#include <stdio.h>
int main() {
    int i, term = 1, diff = 1;
printf("15 terms of the pattern: ");
for (i = 1; i <= 15; i++) {
        printf("%d ", term);
        term += diff;  // Add the current difference
        diff++;       // Increment the difference for the next term
    }
printf("\n");
    return 0;
}
```

20.Write a C Program to print even and odd number from an array

```
#include <stdio.h>
int main() {
    int arr[] = {10, 25, 13, 17, 8, 4};  // Example array
    int n = sz(arr) / sz(arr[0]);
 printf("Even numbers: ");
    for (int i = 0; i < n; i++) {
        if (arr[i] % 2 == 0) {
            printf("%d ", arr[i]);
        }
    }
printf("\nOdd numbers: ");
    for (int i = 0; i < n; i++) {
        if (arr[i] % 2 != 0) {
            printf("%d ", arr[i]);
        }
    }
 printf("\n");
    return 0;
}
```

21.Write a C Program to read character from keyboard and display message whether character is alphabet , digit or special symbol.

```
#include <stdio.h>
int main() {
    char ch;
printf("Enter a character: ");
    scanf("%c", &ch);
 if ((ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z')) {
        printf("%c is an alphabet.\n", ch);
    } else
if (ch >= '0' && ch <= '9') {
        printf("%c is a digit.\n", ch);
    } else {
        printf("%c is a special symbol.\n", ch);
    }
return 0;
}
```

22.Write a C Program to read a string and count number of vowels in it.

```
#include<stdio.h>
int main() {
    char str[100];
    int i, vowels = 0;
printf("Enter a string: ");
    fgets(str, 100, stdin);
for (i = 0; str[i] != '\0'; i++) {
        if (str[i] == 'a' || str[i] == 'e' || str[i] == 'i' || str[i] == 'o' || str[i] == 'u' ||
            str[i] == 'A' || str[i] == 'E' || str[i] == 'I' || str[i] == 'O' || str[i] == 'U') {
            vowels++;
        }
    }
 printf("Number of vowels in the string: %d\n", vowels);
    return 0;
}
```

23.Write a C language program to display the largest element in the matrix.

```
#include <stdio.h>
int main() {
int m, n, i, j, largest = 0;
    scanf("%d %d", &m, &n);
 int matrix[m][n];
printf("Enter the elements of the matrix:\n");
    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j++) {
            scanf("%d", &matrix[i][j]);
            if (matrix[i][j] > largest) {
                largest = matrix[i][j];
            }
        }
    }

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

24.Write a C language program to sp two numbers using pointers and function.

```
#include <stdio.h>
void sp(int *x, int *y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}
int main() {
    int a = 10, b = 20;
printf("Before spping: a = %d, b = %d\n", a, b);
    sp(&a, &b);  // Pass addresses of a and b
    printf("After spping: a = %d, b = %d\n", a, b);
return 0;
}
```
25.Write a C language program to calculate the series- 1/1! + 2/2! + 3/3! + ... Up to n terms.
```
#include <stdio.h>
long long fact(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * fact(n - 1);
    }
}
int main() {
    int n, i;
float sum = 0.0;
 printf("Enter the value of n: ");
    scanf("%d", &n);
 for (i = 1; i <= n; i++) {
        sum += (float)i / fact(i);
    }
 printf("Sum of the series = %.4f\n", sum);
    return 0;
}
```
26. Write down C language program to find out number of occurrences of a character in a file.
```
#include <stdio.h>
int main() {
    FILE *fp;
    char ch, filename[50], search_char;
    int count = 0;
printf("Enter the filename: ");
    scanf("%s", filename);
printf("Enter the character to search: ");
    scanf(" %c", &search_char);
 fp = fopen(filename, "r");
    if (fp == NULL) {
        printf("Error opening file!\n");
        return 1;
    }
 while ((ch = fgetc(fp)) != EOF) {
        if (ch == search_char) {
            count++;
        }
    }
fclose(fp);
    printf("%c occurs %d times in the file.\n", search_char, count);
    return 0;
}
```
27. Write a C language program to display the student result sheet using the data stored in a file.

Student structure
Name character(25)
Rollno integer
Marks1 integer
Marks2 integer
Marks3 integer

```
#include <stdio.h>
struct Student {
    char name[25];
    int rollno;
    int marks1, marks2, marks3;
};
int main() {
    FILE *fp;
    struct Student student;

    fp = fopen("student_data.txt", "r");
    if (fp == NULL) {
        printf("Error opening file!\n");
        return 1;
    }
 printf("\nStudent Result Sheet\n");
    printf("--------------------\n");
while (fread(&student, sz(struct Student), 1, fp) == 1) {
        printf("Name: %s\n", student.name);
        printf("Roll No: %d\n", student.rollno);
        printf("Marks 1: %d\n", student.marks1);
        printf("Marks 2: %d\n", student.marks2);
        printf("Marks 3: %d\n", student.marks3);
        printf("Total: %d\n\n", student.marks1 + student.marks2 + student.marks3);
    }
fclose(fp);
    return 0;
}
```

28.Write a C language program to find out sum of the following series 1! +2! + 3!+. . . ..+n!

```
#include <stdio.h>
long long fact(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * fact(n - 1);
    }
}
int main() {
    int n;
    long long sum = 0;
 printf("Enter the value of n: ");
    scanf("%d", &n);
for (int i = 1; i <= n; i++) {
        sum += fact(i);
    }
printf("Sum of the series 1! + 2! + ... + n! = %lld\n", sum);
    return 0;
}
```

29. Write a C language program to enter n elements in array and find second smallest number from an array.

```
#include <stdio.h>
int main() {
    int n, i, arr[100], first, second;

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

    printf("Enter the elements:\n");
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
 // Assume the first two elements are the smallest initially
    first = second = arr[0];
for (i = 1; i < n; i++) {
        if (arr[i] < first) {
            second = first;
            first = arr[i];
        } else if (arr[i] < second && arr[i] != first) {
            second = arr[i];
        }
    }
 if (second == first) {  // If all elements are the same
        printf("All elements are equal.\n");
    } else {
        printf("Second smallest element: %d\n", second);
    }

    return 0;
}
```

30. Write a C language program to check whether given number is prime or not.

```
#include <stdio.h>
int main() {
    int num, i, isPrime = 1;
printf("Enter a positive integer: ");
    scanf("%d", &num);
 if (num <= 1) {
        isPrime = 0;
    } else {
        for (i = 2; i * i <= num; i++) {
            if (num % i == 0) {
                isPrime = 0;
                break;
            }
        }
    }
 if (isPrime) {
        printf("%d is a prime number.\n", num);
    } else {
        printf("%d is not a prime number.\n", num);
    }

    return 0;
}
```

31. Write a C language program using recursive function to enter 4 digit number and find the sum of all digits of the number .

```
#include <stdio.h>
int sumofdg(int num) {
    if (num == 0) {
        return 0;
    } else {
        return (num % 10) + sumofdg(num / 10);
    }
}
int main() {
    int num;
printf("Enter a 4-digit number: ");
scanf("%d", &num);
if (num >= 1000 && num <= 9999) {
        int sum = sumofdg(num);
        printf("Sum of digits: %d\n", sum);
    } else {
        printf("Invalid input: Please enter a 4-digit number.\n");
    }
 return 0;
}
```

32.Write a C language program to print all Armstrong numbers between 1 to 500 (e.g.153=13+53+33=153)

```
#include <stdio.h>
#include <math.h>
int armstrong(int num) {
    int original = num, rm, n = 0, result = 0;
    while (original != 0) {
        original /= 10;
        ++n;
    }
original = num;

    while (original != 0) {
        rm = original % 10;
        result += pow(rm, n);
        original /= 10;
    }
 return (result == num);  // True if Armstrong, false otherwise
}
int main() {
    for (int i = 1; i <= 500; i++) {
        if (armstrong(i)) {
            printf("%d ", i);
        }
    }
    printf("\n");
    return 0;
}
```

33.Write a C language program to find whether given number is palindrome or not.

```
#include <stdio.h>
int pld(int num) {
    int reversed = 0, rm, original = num;
 while (num != 0) {
        rm = num % 10;
        reversed = reversed * 10 + rm;
        num /= 10;
    }
return (original == reversed);
}
int main() {
    int num;
printf("Enter an integer: ");
    scanf("%d", &num);
 if (pld(num)) {
        printf("%d is a palindrome.\n", num);
    } else {
        printf("%d is not a palindrome.\n", num);
    }
return 0;
}
```

34.Write a C language program to find GCD of given two numbers.

```
#include <stdio.h>
int gcd(int a, int b) {
    while (b != 0) {
        int temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}
int main() {
    int n1, n2;
 printf("Enter two integers: ");
    scanf("%d %d", &n1, &n2);

    int gcd_value = gcd(n1, n2);
    printf("GCD of %d and %d is %d\n", n1, n2, gcd_value);
return 0;
}
```

35. Write a C language program which will read string and count the number of characters and words in it.

```
#include <stdio.h>
int main() {
    char str[100];
    int i, chars = 0, words = 1;  // Start with 1 word for the first word
printf("Enter a string: ");
    fgets(str, 100, stdin);
for (i = 0; str[i] != '\0'; i++) {
        chars++;
        if (str[i] == ' ' || str[i] == '\n' || str[i] == '\t') {
            words++;
        }
    }
 printf("%d\n", chars);
    printf("%d\n", words);
 return 0;
}
```

36. Write a C language program to read two matrices and add them.

```
#include <stdio.h>
int main() {
    int m, n, i, j;
scanf("%d %d", &m, &n);
  int matrix1[m][n], matrix2[m][n], sum[m][n];
    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j++) {
            scanf("%d", &matrix1[i][j]);
        }
    }
printf("Enter elements of matrix 2:\n");
    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j++) {
            scanf("%d", &matrix2[i][j]);
        }
    }
    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j++) {
            sum[i][j] = matrix1[i][j] + matrix2[i][j];
        }
    }
printf("Sum of the matrices:\n");
    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j++) {
            printf("%d ", sum[i][j]);
        }
        printf("\n");
    }
 return 0;
}
```

37. Write a C language program to read two matrices and multiply them.

```
#include <stdio.h>
int main() {
    int m, n, p, q, i, j, k;
    scanf("%d %d", &m, &n);
    scanf("%d %d", &p, &q);
 if (n != p) {
        printf("Invalid dimensions for multiplication!\n");
        return 1;
    }
int matrix1[m][n], matrix2[p][q], product[m][q];
printf("Enter elements of matrix 1:\n");
    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j++) {
            scanf("%d", &matrix1[i][j]);
        }
    }
 printf("Enter elements of matrix 2:\n");
    for (i = 0; i < p; i++) {
        for (j = 0; j < q; j++) {
            scanf("%d", &matrix2[i][j]);
        }
    }
// Multiply the matrices
    for (i = 0; i < m; i++) {
        for (j = 0; j < q; j++) {
            product[i][j] = 0;
            for (k = 0; k < n; k++) {
                product[i][j] += matrix1[i][k] * matrix2[k][j];
            }
        }
    }
printf("Product of the matrices:\n");
    for (i = 0; i < m; i++) {
        for (j = 0; j < q; j++) {
            printf("%d ", product[i][j]);
        }
        printf("\n");
    }
 return 0;
}
```

38.Write a C language program to read one matrix and find the sum of it’s diagonal elements.

```
#include <stdio.h>
int main() {
    int m, n, i, j, sum = 0;
    scanf("%d %d", &m, &n);
int matrix[m][n];
 printf("Enter elements of the matrix:\n");
    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j++) {
            scanf("%d", &matrix[i][j]);
        }
    }

    for (i = 0; i < m && i < n; i++) {
        sum += matrix[i][i];
    }
printf("Sum of diagonal elements: %d\n", sum);
return 0;
}
```

39. Write a C language program to input number and find a largest digit in a given number and print it in word with appropriate message. (e.g.n=5273 - “SEVEN is largest”)

```
#include <stdio.h>
char *digits[] = {"ZERO", "ONE", "TWO", "THREE", "FOUR", "FIVE", "SIX", "SEVEN", "EIGHT",
"NINE"};
int main() {
    int num, largest = 0, digit;
    scanf("%d", &num);

while (num > 0) {
        digit = num % 10;
        if (digit > largest) {
            largest = digit;
        }
        num /= 10;
    }
printf("%s ", digits[largest]);
 return 0;
}
```

40. Write a C language program to compute following series G= 1+ x3/3! + x5 /5! + x7/7! + ... up to n terms.

```
#include <stdio.h>
int fact(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * fact(n - 1);
    }
}
double compute_series(int x, int n) {
    double sum = 1;
    for (int i = 1; i <= n; i += 2) {
        sum += (double)pow(x, i) / fact(i);
    }
    return sum;
}
int main() {
    int x, n;
printf("Enter the value of x: ");
    scanf("%d", &x);
 printf("Enter the number of terms: ");
    scanf("%d", &n);
double result = compute_series(x, n);
    printf("Sum of the series G = %.2lf\n", result);
 return 0;
}
```

41. Write a C language program to read n numbers in an array and split the array into two arrays even and odd such that the array even contains all the even numbers and other is odd. So the output will be—(e.g. Original array is 7,9,4,6,5,3,2,10,18 Odd array is 7,9,5,3 Even array is 4,6,2,10,18)

```
#include <stdio.h>
int main() {
    int n, i, count= 0, odd_count = 0;
 printf("Enter the number of elements: ");
    scanf("%d", &n);
int arr[n], even[n], odd[n];
printf("Enter the elements:\n");
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
        if (arr[i] % 2 == 0) {
            even[even_count++] = arr[i];
        } else {
            odd[odd_count++] = arr[i];
        }
    }
printf("Original array: ");
    for (i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
printf("\nOdd array: ");
    for (i = 0; i < odd_count; i++) {
        printf("%d ", odd[i]);
    }
 printf("\nEven array: ");
    for (i = 0; i < even_count; i++) {
        printf("%d ", even[i]);
    }
 printf("\n");
    return 0;
}
```

42.Write a C language program to check whether the string is palindrome or not.

```
#include <stdio.h>
#include <string.h>
int pld(char str[]) {
    int len = strlen(str);
    for (int i = 0; i < len / 2; i++) {
        if (str[i] != str[len - i - 1]) {
            return 0;
        }
    }
    return 1;
}
int main() {
    char str[100];
printf("Enter a string: ");
    fgets(str, 100, stdin);
    str[strcspn(str, "\n")] = '\0'

    if (pld(str)) {
        printf("%s is a palindrome.\n", str);
    } else {
        printf("%s is not a palindrome.\n", str);
    }

    return 0;
}
```

43.Write a C language program to add, list, delete record and modify the current record.

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

struct Record {
    int id;
    char name[50];
    int salary;
};

// Function prototypes for record operations
void add_record(struct Record records[], int *count);
void list_records(struct Record records[], int count);
void delete_record(struct Record records[], int *count);
void modify_record(struct Record records[], int count);

int main() {
    struct Record records[10];
    int count = 0;
    int choice;
  do {
        printf("\nMenu:\n");
        printf("1. Add record\n");
        printf("2. List records\n");
        printf("3. Delete record\n");
        printf("4. Modify record\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
 switch (choice) {
            case 1:
                add_record(records, &count);
                break;
            case 2:
                list_records(records, count);
                break;
            case 3:
                delete_record(records, &count);
                break;
            case 4:
                modify_record(records, count);
                break;
            case 5:
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid choice!\n");
        }
    } while (choice != 5);
return 0;
}
```

44. Write a C language program using structure to define employee record containing employee number, name and salary. Read 10 records.

```
#include <stdio.h>
struct Employee {
    int emp_num;
    char name[50];
    float salary;
};
int main() {
    struct Employee emp[10];
 printf("Enter details of 10 employees:\n");
    for (int i = 0; i < 10; i++) {
        printf("Employee %d:\n", i + 1);
        printf("Enter employee number: ");
        scanf("%d", &emp[i].emp_num);
        printf("Enter name: ");
        fgets(emp[i].name, 50, stdin);
        emp[i].name[strcspn(emp[i].name, "\n")] = '\0';  // Remove trailing newline
        printf("Enter salary: ");
        scanf("%f", &emp[i].salary);
    }
printf("\nEmployee details:\n");
    for (int i = 0; i < 10; i++) {
        printf("Employee %d:\n", i + 1);
        printf("Employee number: %d\n", emp[i].emp_num);
        printf("Name: %s\n", emp[i].name);
        printf("Salary: %.2f\n", emp[i].salary);
    }
 Return 0;
}
```

44.Write a C language program to demonstrate the use of union.

```
#include <stdio.h>
union Data {
    int i;
    float f;
    char str[20];
};
int main() {
    union Data data;
    data.i = 10;
    printf("Integer value: %d\n", data.i);
    data.f = 3.14;
    printf("Float value: %f\n", data.f);
    strcpy(data.str, "Hello, world!");
    printf("String value: %s\n", data.str);
return 0;
}
```

46. Write a C language program to define structure for class containing class, name, no. of students and block no. Read 5 records and display it.

```
#include <stdio.h>
struct Class {
    char name[50];
    int nmstud;
    int block_no;
};
int main() {
    struct Class classes[5];
printf("Enter details of 5 classes:\n");
    for (int i = 0; i < 5; i++) {
        printf("Class %d:\n", i + 1);
        fgets(classes[i].name, 50, stdin);
        classes[i].name[strcspn(classes[i].name, "\n")] = '\0';
        scanf("%d", &classes[i].nmstud);
        printf("Enter block number: ");
        scanf("%d", &classes[i].block_no);
    }
printf("\nClass details:\n");
    for (int i = 0; i < 5; i++) {
        printf("Class %d:\n", i + 1);
        printf("Name: %s\n", classes[i].name);
        printf("Number of students: %d\n", classes[i].nmstud);
        printf("Block number: %d\n", classes[i].block_no);
}
  return 0;
}
```

47.Write a C language program using recursion n terms of Fs series.

```
#include <stdio.h>
int fs(int n) {
    if (n <= 1) {
        return n;
    } else {
        return fs(n - 1) + fs(n - 2);
    }
}
Int main()
 {
    int n, i;
printf("Enter the number of terms: ");
    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        printf("%d ", fs(i));
    }
 printf("\n");
    return 0;
}
```

48.Write a C language program using recursion to calculate fact of given number.

```
#include <stdio.h>
int fact(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * fact(n - 1);
    }
}
int main() {
    int num;
printf("Enter a non-negative integer: ");
    scanf("%d", &num);
if (num < 0) {
        printf("Fact is not defined for negative numbers.\n");
    } else {
        long long fact = fact(num);
        printf("Fact of %d = %lld\n", num, fact);
    }

    return 0;
}
```

49.Distinguish between character constant and string constant.

Character Constant:

Represents a single character. Enclosed in single quotes ('). Example: 'A'.

String Constant:

- Represents a sequence of characters.
- Enclosed in double quotes (").

- Example: "Hello".
- Ends with a null character ('\0') to indicate the end of the string.
- Stored as an array of characters.

50. Describe all operators used in C language with example.

Ans:

1. Arithmetic Operators:

- Perform basic arithmetic operations.
- Examples:
  - + (addition): x + y
  - - (subtraction): x - y
  - * (multiplication): x * y
  - / (division): x / y
  - % (modulo): x % y (remainder of x divided by y)

2. Relational Operators:

- Compare two values and return a boolean result (true or false).
- Examples:
  - == (equal to): x == y
  - != (not equal to): x != y
  - > (greater than): x > y
  - < (less than): x < y
  - >= (greater than or equal to): x >= y
  - <= (less than or equal to): x <= y

3. Logical Operators:

- Combine multiple boolean expressions.
- Examples:
  - && (logical AND): x && y (true only if both x and y are true)
  - || (logical OR): x || y (true if either x or y is true)
  - ! (logical NOT): !x (inverts the boolean value of x)

4. Assignment Operators:

- Assign a value to a variable.
- Examples:

  - = (assignment): x = y (assigns the value of y to x)
  - += (add and assign): x += y (equivalent to x = x + y)
  - -= (subtract and assign): x -= y
  - *= (multiply and assign): x *= y
  - /= (divide and assign): x /= y
  - %= (modulo and assign): x %= y

5. Increment and Decrement Operators:

- Increase or decrease a variable's value by 1.
- Examples:
  - ++ (increment): x++ (increments x after using its value) or ++x (increments x before using its value)
  - -- (decrement): x-- (decrements x after using its value) or --x (decrements x before using its value)

6. Bitwise Operators:

- Perform operations on individual bits of integer values.

- Examples:
    - & (bitwise AND): x & y (performs AND operation on each bit of x and y)
    - | (bitwise OR): x | y
    - ^ (bitwise XOR): x ^ y
    - ~ (bitwise NOT): ~x (inverts all bits of x)
    - << (left shift): x << y (shifts x's bits to the left by y positions)
    - >> (right shift): x >> y (shifts x's bits to the right by y positions)

7. Conditional Operator:

- Provides a shorthand for if-else statements.
- Example: condition ? expression1 : expression2 (evaluates expression1 if condition is true, otherwise evaluates expression2)

8. Special Operators:

- sizeof: Returns the size of a variable or data type in bytes.
- &: Address-of operator, returns the memory address of a variable.
- *: Dereference operator, accesses the value at a memory address.
- .: Member access operator, used to access members of structures and unions.
- ->: Arrow operator, used to access members of structures and unions through pointers.50. Describe all operators used in C language with example.

Ans:

1. Arithmetic Operators:

- Perform basic arithmetic operations.
- Examples:
    - + (addition): x + y
    - - (subtraction): x - y
    - * (multiplication): x * y
    - / (division): x / y
    - % (modulo): x % y (remainder of x divided by y)

2. Relational Operators:

- Compare two values and return a boolean result (true or false).
- Examples:
    - == (equal to): x == y
    - != (not equal to): x != y
    - > (greater than): x > y
    - < (less than): x < y
    - >= (greater than or equal to): x >= y
    - <= (less than or equal to): x <= y

3. Logical Operators:

- Combine multiple boolean expressions.
- Examples:

- && (logical AND): x && y (true only if both x and y are true)
- || (logical OR): x || y (true if either x or y is true)
- ! (logical NOT): !x (inverts the boolean value of x)

4. Assignment Operators:

- Assign a value to a variable.
- Examples:

- = (assignment): x = y (assigns the value of y to x)
- += (add and assign): x += y (equivalent to x = x + y)
- -= (subtract and assign): x -= y
- *= (multiply and assign): x *= y
- /= (divide and assign): x /= y
- %= (modulo and assign): x %= y

5. Increment and Decrement Operators:

- Increase or decrease a variable's value by 1.
- Examples:
- ++ (increment): x++ (increments x after using its value) or ++x (increments x before using its value)
- -- (decrement): x-- (decrements x after using its value) or --x (decrements x before using its value)

6. Bitwise Operators:

- Perform operations on individual bits of integer values.
- Examples:
- & (bitwise AND): x & y (performs AND operation on each bit of x and y)
- | (bitwise OR): x | y
- ^ (bitwise XOR): x ^ y
- ~ (bitwise NOT): ~x (inverts all bits of x)
- << (left shift): x << y (shifts x's bits to the left by y positions)
- >> (right shift): x >> y (shifts x's bits to the right by y positions)

7. Conditional Operator:

- Provides a shorthand for if-else statements.
- Example: condition ? expression1 : expression2 (evaluates expression1 if condition is true, otherwise evaluates expression2)

8. Special Operators:

- sizeof: Returns the size of a variable or data type in bytes.
- &: Address-of operator, returns the memory address of a variable.
- *: Dereference operator, accesses the value at a memory address.
- .: Member access operator, used to access members of structures and unions.
- ->: Arrow operator, used to access members of structures and unions through pointers.

51. Explain in detail three parts of C program.

1.Header file:
The header files in a C program contain declarations for functions that are used in the program. They provide essential information to the compiler about the functions, variables, and constants used in the program.It usually contains information about the standard input/output functions and other libraries that the program will use.

Example:

```
#include <stdio.h>   // Standard Input/Output functions #include <stdlib.h> // Standard Library
functions #include <math.h>         // Math functions
```

2. Main Function:
The main() function is the entry point of a C program. It is where the program execution begins. All C programs must have a main() function, and the execution of the program starts from the first statement within this function.

Example:

```
#include <stdio.h> int main() {
printf("Hello, World!\n");
return 0;
}
```

The int before main() indicates that the function returns an integer value to the operating system (0 typically indicates success.

3. Functions:
This section includes user-defined functions or procedures that are used to modularize the code. Functions help in organizing code, making it more readable, maintainable, and reusable.

Example:

```
#include <stdio.h>

// Function declaration int add(int a, int b);

int main() {
int result = add(5, 7); printf("Sum: %d\n", result); return 0;
}

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

52. Write a short note on precedence and order of Evolution.

Ans:

Precedence and Order of Evolution: A Short Note

Evolution isn't a linear process with a strict hierarchy of forces, but rather a complex interplay of several factors influencing its direction and pace. However, understanding the relative importance of these factors helps us comprehend the fascinating journey of life on Earth. Here's a simplified view of the precedence and order of evolution:

1. Natural Selection: The primary driver, favoring individuals with traits better suited to their environment for survival and reproduction. Think camouflage in animals, antibiotic resistance in bacteria.

2. Genetic Drift: Random fluctuations in allele frequencies within populations, independent of selection pressure. Can be significant in small populations, contributing to new species formations.

3. Gene Flow: Movement of genes between populations through breeding or migration. Introduces new genetic variations and alters allele frequencies. Think the spread of insecticide resistance in

insects.

4. Mutations: Random changes in the genetic material. Can be beneficial, detrimental, or neutral, but provide the raw material for selection and other forces to act upon. Consider antibiotic resistance in bacteria, lactose tolerance in humans.

5. Environmental Fluctuations: Changes in the surrounding environment impose new selection pressures, influencing the direction of evolution. Think mass extinctions driving diversification, climate change impacting species distribution.

Points to Consider:

- This order is not absolute, and the interplay between these factors can be intricate and context-dependent.
- Epigenetics and developmental factors can also play significant roles.
- Evolution operates over vast timescales, and its outcomes are often unpredictable.

52.Write a short note on precedence and order of evolution.

Bitwise operators are operators in C that work on the individual bits of integer values. They are used for tasks like:

1. Bitwise AND (&):

- Compares each corresponding pair of bits in two operands.
- Sets the result bit to 1 only if both bits in the operands are 1.
- Example: 10 & 5 = 0 (binary: 1010 & 0101 = 0000)

2. Bitwise OR (|):

- Sets the result bit to 1 if either or both bits in the operands are 1.
- Example: 10 | 5 = 15 (binary: 1010 | 0101 = 1111)

3. Bitwise XOR (^):

- Sets the result bit to 1 if the corresponding bits in the operands are different.

- Example: 10 ^ 5 = 15 (binary: 1010 ^ 0101 = 1111)

4. Bitwise NOT (~):

- Inverts all the bits of a single operand.
- Example: ~10 = -11 (binary: ~00001010 = 11110101)

5. Bitwise Left Shift (<<):

- Shifts the bits of the first operand to the left by the number of positions specified by the second operand.
- Example: 8 << 2 = 32 (binary: 00001000 << 2 = 00100000)

6. Bitwise Right Shift (>>):

- Shifts the bits of the first operand to the right by the number of positions specified by the second operand.
- Example: 16 >> 1 = 8 (binary: 00010000 >> 1 = 00001000)

Common Applications of Bitwise Operators:

- Toggling bits: x ^= (1 << n) toggles the nth bit of x.
- Setting bits: x |= (1 << n) sets the nth bit of x to 1.
- Clearing bits: x &= ~(1 << n) clears the nth bit of x to 0.
- Checking bits: if (x & (1 << n)) checks if the nth bit of x is 1.
- Masking: Isolating specific bits or groups of bits.
- Data compression: Compressing data by storing multiple values in a single integer.
- Cryptography: Implementing encryption and decryption algorithms.
- Image processing: Manipulating pixels at the bit level.
- Low-level hardware programming: Interacting with hardware registers and memory.

54. State and explain formatted input-output statements and standard input-output statements with example.

Here's an explanation of formatted and standard input-output statements in C, with examples:
Formatted Input-Output Statements:
Purpose:

- Control the appearance of input and output data for better readability and presentation.

Statements:

- printf(): Used for formatted output.
- scanf(): Used for formatted input.

Working:

- Use format specifiers (e.g., %d, %f, %c, %s) within a format string to specify how to interpret or display data.

Example:

```
// Formatted output:
int age = 25;
float salary = 1234.56;
printf("Age: %d\nSalary: $%.2f\n", age, salary);
```

```
// Output: Age: 25
//        Salary: $1234.56

// Formatted input:
int num; float price;
printf("Enter a number: "); scanf("%d", &num); printf("Enter a price: "); scanf("%f", &price);
```

Standard Input-Output Statements:

Purpose:

- Handle input and output without specific formatting for basic interactions.

Statements:

- getchar(): Reads a single character from standard input (stdin).
- putchar(): Prints a single character to standard output (stdout).

Example:

C
```
// Input a character:
char ch = getchar();

// Output a character:
putchar('A');
```

55.Explain in detail call by value and call by reference with example.

Call by Value:
When a function is called, a copy of the argument's value is passed to the function.
Changes made to the argument within the function do not affect the original value in the calling program.
Example:
```
void sp_by_value(int x, int y) {
   int temp = x;
x = y;
 y = temp;
}
int main() {
   int a = 10, b = 20;
   sp_by_value(a, b);  // Does not affect a and b in main
   printf("a = %d, b = %d\n", a, b);  // Output: a = 10, b = 20
```
Call by Reference:
The address of the argument is passed to the function.
The function can directly modify the original value in the calling program.
Use the & operator to pass arguments by reference.
Example:
```
void sp_by_reference(int *x, int *y) {
   int temp = *x;
   *x = *y;
```

```
    *y = temp;
}
int main() {
    int a = 10, b = 20;
    sp_by_reference(&a, &b);  // Affects a and b in main
    printf("a = %d, b = %d\n", a, b);  // Output: a = 20, b = 10
}
```

56.Explain the following using general syntax and example . i)if ii) if-else iii) nested if-else

i) if statement:
Syntax:

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

Example:

```
int age = 25;
if (age >= 18) {
    printf("You are eligible to vote.\n");
}
```

ii) if-else statement:
Syntax:

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

Example:

```
int number = 10;
if (number % 2 == 0) {
    printf("The number is even.\n");
} else {
    printf("The number is odd.\n");
}
```

iii) Nested if-else statement:
Syntax:

```
if (condition1) {
    // statements to execute if condition1 is true
    if (condition2) {
        // statements to execute if condition2 is also true
    } else {
        // statements to execute if condition2 is false
    }
} else {
    // statements to execute if condition1 is false
}
```

Example:

```
int grade = 85;
if (grade >= 90) {
    printf("Excellent!\n");
} else if (grade >= 80) {
    printf("Very good!\n");
} else {
    printf("Good job. Keep working hard!\n");
}
```

57.Explain break and continue statements using syntax and example .

1.Break statement:

Immediately terminates the innermost loop or switch statement.

Syntax:

```
break;
```

Example:

```
for (int i = 1; i <= 10; i++) {
    if (i == 5) {
        break;  // Exit the loop when i becomes 5
    }
    printf("%d ", i);
}
```

2.Continue statement:

Skips the remaining statements in the current iteration of a loop and jumps to the next iteration.

Syntax:

```
continue;
```

Example:

```
for (int i = 1; i <= 10; i++) {
    if (i % 2 == 0) {
        continue;  // Skip even numbers
    }
    printf("%d ", i);  // Print odd numbers
}
```

58.Explain following ,i)while ii) do-while iii) for

i) while loop:

Repeats a block of code as long as a specified condition is true.

Syntax:

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

Example:

```
int i = 1;
while (i <= 5) {
    printf("%d ", i);
    i++;
}
```

ii) do-while loop:

Executes a block of code at least once, then repeats it as long as a condition is true.

Syntax:

```
do {
    // statements to execute at least once
} while (condition);
```

Example:

```
int choice;
do {
    printf("Enter a number (0 to quit): ");
    scanf("%d", &choice);
```

} while (choice != 0);
iii) for loop:
Repeats a block of code a specific number of times.
Syntax:
for (initialization; condition; increment/decrement) {
59.Define array. Explain different types of array in detail.
Ans:

Here's a comprehensive explanation of arrays in C:

Definition:

- An array is a collection of elements of the same data type, stored in contiguous memory locations under a common name.
- Each element can be accessed directly using its index, which starts from 0.

Types of Arrays:

1. One-Dimensional Arrays:
    - Simplest type, containing a single row of elements.
    - Declaration: `data_type array_name[size];`
    - Example: `int numbers[5];`
2. Multidimensional Arrays:
    - Represent data in multiple dimensions, like rows and columns.
    - Two-Dimensional Arrays:
        - Represent data in a table-like structure with rows and columns.
        - Declaration: `data_type array_name[rows][columns];`
        - Example: `int matrix[3][4];`

    - Higher-Dimensional Arrays:
        - Can have more than two dimensions, although less common.
3. Arrays as Function Arguments:
    - Passed to functions by reference (address of the first element is passed).
    - Changes made within the function affect the original array.
4. Character Arrays:
    - Used to store strings (sequences of characters).
    - Terminated by a null character (`\0`).
5. Dynamic Arrays:
    - Allocated at runtime using `malloc()` or `calloc()`.
    - Size can be adjusted as needed.

60. State and explain various types of standard function with example.

printf(): This function is used to print formatted output. Example:
printf("Hello, %s!\n", "world");
scanf(): It is used for reading input. Example:
int num; scanf("%d", &num);
strlen(): Calculates the length of a string.

char str[] = "Hello";
int length = strlen(str);

strcpy(): Copies one string to another. Example:

char source[] = "Copy me!"; char destination[20]; strcpy(destination, source);

atoi(): Converts a string to an integer. Example:

char numStr[] = "123"; int num = atoi(numStr);

rand(): Generates a pseudo-random number. Example:

int randomNum = rand() % 100;   // Generates a random number between 0 and 99

malloc() and free(): Used for dynamic memory allocation and deallocation. Example:

int *arr = (int*)malloc(5 * sizeof(int));
// ... use arr ... free(arr);

strcmp(): Compares two strings. Example:

char str1[] = "hello"; char str2[] = "world";
int result = strcmp(str1, str2);:

59. Arrays in C:

Definition:

An array is a collection of elements of the same data type, stored in contiguous memory locations under a common name.
Each element is accessed using an index, starting from 0.
Different Types of Arrays:

One-Dimensional Arrays:

Simplest form of arrays.
Declare using: data_type array_name[size];
Example: int numbers[5];
Multidimensional Arrays:

Arrays with multiple dimensions (like matrices).
Declare using: data_type array_name[size1][size2]...[sizeN];
Example: int matrix[3][4];
Character Arrays (Strings):

Arrays of characters, used to store text.
Declare using: char string_name[size];
Terminated by a null character (\0).
Arrays of Pointers:

Arrays that store pointers to other variables.
Declare using: data_type *array_name[size];
Key Points:

Arrays have a fixed size that cannot be changed after declaration.
Array elements can be accessed, modified, and iterated through using loops.
Arrays are often used to store and manage collections of data efficiently.
60. Standard Functions in C:
Types of Standard Functions:
1.Input/Output Functions:
printf(): Formatted output to console.
scanf(): Formatted input from console.
getchar(): Read a single character.
putchar(): Print a single character.
2.Mathematical Functions:
sqrt(): Square root.
pow(): Raise a number to a power.
sin(), cos(), tan(): Trigonometric functions.
abs(): Absolute value.
61.State and explain different phases used in user defined function.
1. Function Declaration:
Introduces the function's name, return type, and parameters to the compiler.
Syntax: return_type function_name(parameters);
Example: int add(int a, int b);
2. Function Definition:
Contains the actual code to be executed when the function is called.
Syntax:
```
return_type function_name(parameters) {
    // Function body (statements)
}
```
Example:
```
int add(int a, int b) {
    return a + b;
}
```
62. Explain function with return and function with arguments with example.

1.Functions with Return:

Return a value to the calling code using the return statement.

Syntax:
```
return_type function_name(parameters) {
    // Function body
    return value;
}
```
Example:
```
int calculate_area(int length, int width) {
    return length * width;
}
```
2.Functions with Arguments:

Accept values (arguments) from the calling code when they are called.

Parameters are used to hold these arguments within the function.

Syntax:
```
return_type function_name(parameter_list) {
    // Function body
}
```
Example:
```
void greet(char name[]) {
    printf("Hello, %s!\n", name);
}
```
63. State and explain different types of string functions with example.

1. Length Determination:

strlen(str): Returns the length of the string str (excluding the null terminator). Example: int len = strlen("Hello"); // len will be 5

2. String Copying:

strcpy(dest, src): Copies the string src to dest. Example: strcpy(name2, name1);

3. String Concatenation:

strcat(dest, src): Appends the string src to the end of dest. Example: strcat(message, " World");

64. Explain dynamic memory allocation and releasing dynamically allocated memory.

Dynamic Memory Allocation:

Allows you to allocate memory during program execution, as needed.

Key functions:

malloc(size): Allocates a block of size bytes of memory. Returns a void pointer to the allocated memory.

calloc(num_elements, element_size): Allocates memory for an array of num_elements, each of element_size bytes. Initializes the memory to zero.

realloc(ptr, new_size): Resizes a previously allocated block of memory pointed to by ptr to a new size of new_size bytes.

Releasing Dynamically Allocated Memory:

Crucial to prevent memory leaks.

Use free(ptr) to release memory previously allocated with malloc, calloc, or realloc.

Example:

```
int *numbers = (int *)malloc(5 * sz(int));
// ... use the memory ...
free(numbers);  // Release the memory
```

65.Define structure and union. Explain the way of declaring and accessing them.

Structures:

Definition: A user-defined composite data type that groups variables of different data types under a single name.

Declaration:

```
struct structure_name {
    data_type member1;
    data_type member2;
    // ... more members
};
```

Accessing Members:

```
struct_variable.member_name
```

Example:

```
struct Student {
    int roll_no;
    char name[50];
    float marks;
};
struct Student student1;
student1.roll_no = 10;
strcpy(student1.name, "Alice");
student1.marks = 85.5;
```

Unions:

Definition: A user-defined data type that allows storing different data types in the same memory location, but only one member can hold a value at a time.

Declaration:

```
union union_name {
    data_type member1;
    data_type member2;
    // ... more members
};
```

Accessing Members:

```
union_variable.member_name
```

Example:

```
union Data {
    int i;
    float f;
    char str[20];
};
union Data data;
data.i = 10;  // Only i holds a value now
strcpy(data.str, "Hello");  // Overwrites the value of i
```

66. Nested Structures and Self-Referential Structures:

Nested Structures:

Structures that contain other structures as members.

Example:

```
struct Address {
    char city[50];
    char state[50];
};
struct Employee {
    int id;
    char name[50];
    struct Address address;  // Nested structure
};
```
Accessing nested members: employee1.address.city
Self-Referential Structures:
Structures that contain a pointer to the same structure type.
Used for creating linked lists, trees, and other recursive data structures.
Example:
```
struct Node {
    int data;
    struct Node *next;  // Pointer to the next node
};
```
69.What do you mean by pre-processor? Explain in detail macros.
Preprocessor:
A text-processing phase that occurs before actual compilation of a C program.
It handles instructions called preprocessor directives, which start with the # symbol.
Its primary tasks include:
Macro expansion
File inclusion
Conditional compilation
Macros:
Text substitutions defined using the #define directive.
Replace a macro name with its defined value throughout the code before compilation.
Types:
Object-like macros: Simple text replacements (e.g., #define PI 3.14159)
Function-like macros: Simulate functions but lack type checking and argument safety (e.g., #define MX(x, y) ((x) > (y) ? (x) : (y)))
70.What do you mean by pre-processor directives? List and explain its different categories.
Instructions for the preprocessor, beginning with #.
Categories:
1.File Inclusion:
#include – Inserts contents of another file into the current file.
#include <header_file.h> for standard headers
#include "user_header.h" for user-defined headers
2.Macro Definition:
#define – Defines macros for text substitution.
3.Conditional Compilation:
#ifdef, #ifndef, #if, #else, #elif, #endif – Control which code sections are compiled based on conditions.
4.Other Directives:
#undef – Undefines a macro.
#error – Generates an error message.
#pragma – Compiler-specific instructions.
71. Write a C language program to enter n elements in array and find second largest number from
array.

```
#include <stdio.h>
int main() {
    int n, i, largest, second_largest;

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

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

    // Assume the first two elements are the largest and second largest
    if (arr[0] > arr[1]) {
        largest = arr[0];
        second_largest = arr[1];
    } else {
        largest = arr[1];
        second_largest = arr[0];
    }

    // Iterate through the rest of the array
    for (i = 2; i < n; i++) {
        if (arr[i] > largest) {
            second_largest = largest;
            largest = arr[i];
        } else
 if (arr[i] > second_largest && arr[i] != largest) {
            second_largest = arr[i];
        }
}
   printf("Second largest number: %d\n", second_largest);

    return 0;
}
```