Q1: "What are the features of the C programming language?"
Ans: "C is simple, portable, structured, procedural, fast, and allows low-level memory access."

Q2: "Difference between Compiler and interpreter."
Ans: "Compiler translates the whole program at once into machine code, while an interpreter translates and executes line by line."

Q3: "What is the meaning of #include in a C program?"
Ans: "#include is used to include header files or libraries in a program."

Q4: "Difference between #include and #include "filename"."
Ans: " searches in standard libraries, "filename" searches in the current directory first."

Q5: "What are header files and their uses?"
Ans: "Header files contain function declarations, macros, and constants that can be reused across programs."

Q6: "Why is C called a mid-level programming language?"
Ans: "C combines high-level features with low-level memory access capabilities."

Q7: "Use of printf() and scanf() functions; explain format specifiers."
Ans: "printf() outputs data; scanf() takes input. Format specifiers define data type: %d (int), %f (float), %c (char), %s (string)."

Q8: "Use of a semicolon (;) at the end of every statement."
Ans: "Semicolon indicates the end of a statement in C."

Q9: "What is volatile in C?"
Ans: "volatile tells the compiler that a variable's value may change unexpectedly and prevents optimization."

Q10: "What is typecasting in C?"
Ans: "Typecasting converts a variable from one data type to another."

Q11: "What are bitwise operators? Provide examples."
Ans: "Operators that work on bits: AND (&), OR (|), XOR (^), NOT (~), left shift (<<), right shift (>>)."

Q12: "What happens if you don't use a break statement in a switch case?"
Ans: "Execution continues to the next case (fall-through) until a break is encountered or switch ends."

Q13: "Why use default statements in switch cases?"
Ans: "Default handles cases not matched by any explicit case."

Q14: "Explain if, if-else, and else-if statements in C."
Ans: "if checks a condition, if-else provides two paths, else-if allows multiple conditional checks."

Q15: "How do loops work in C? Explain for, while, and do-while loops."
Ans: "Loops repeat code: for (fixed iteration), while (condition first), do-while (executes once before checking)."

Q16: "How can you exit from a loop in C?"
Ans: "Use break to exit the loop or return to exit a function."

Q17: "What are macros in C?"
Ans: "Macros are preprocessor directives defined using #define to replace code before compilation."

Q18: "What are command line arguments?"
Ans: "Arguments passed to a program at runtime via main(int argc, char *argv[])."

Q19: "Difference between macro and functions."
Ans: "Macros are preprocessor substitutions; functions are executed at runtime and have type checking."

Q20: "Difference between call by value and call by reference."
Ans: "Call by value passes a copy of data; call by reference passes the actual address."

Q21: "What is pass-by-reference in functions?"
Ans: "Passing the actual address so changes affect the original variable."

Q22: "What are goto statements? Should we use them?"
Ans: "goto jumps to a labeled statement. It is generally discouraged due to poor readability."

Q23: "How to return array in C?"
Ans: "Cannot return local array directly; use pointers or pass array to function to modify."

Q24: "What is an array in C?"
Ans: "Array is a collection of elements of the same type stored in contiguous memory."

Q25: "How do you declare and initialize an array in C?"
Ans: "int arr[5]; or int arr[5] = {1,2,3,4,5};"

Q26: "Syntax for accessing array elements."
Ans: "Use index: arr[i] accesses the ith element."

Q27: "Difference between one-dimensional and multi-dimensional array."
Ans: "1D array stores in a single row, multi-D array stores data in multiple rows/columns."

Q28: "How to determine the size of an array?"
Ans: "sizeof(arr)/sizeof(arr[0]) gives the number of elements."

Q29: "Program to sort a string of characters."
Ans: "Use nested loops to compare and swap characters using ASCII values."

Q30: "Program to count unique characters in a string."
Ans: "Use an array of size 256 to track occurrence of each character and count unique ones."

Q31: "How to initialize 1D array at declaration?"
Ans: "int arr[5] = {1,2,3,4,5};"

Q32: "How to iterate over elements of 1D array?"
Ans: "Use for loop: for(int i=0;i<size;i++) { /* access arr[i] */ }"

Q33: "C program to find max and min in 1D array."
Ans: "Iterate through array, update max and min variables."

Q34: "How to pass 1D array to a function?"
Ans: "Pass array name: void func(int arr[], int size)."

Q35: "How to return an array from a function?"
Ans: "Return pointer to dynamically allocated array or use static array."

Q36: "Declare and initialize 2D array."
Ans: "int arr[3][3] = {{1,2,3},{4,5,6},{7,8,9}};"

Q37: "Access elements of 2D array."
Ans: "Use arr[i][j] to access row i, column j."

Q38: "Pass 2D array to function."
Ans: "Specify column size: void func(int arr[][3], int rows)."

Q39: "Work with multi-dimensional arrays beyond 2D."
Ans: "Use multiple indices: arr[i][j][k] etc., and pass correct dimensions to functions."

Q40: "Declare and initialize string in C."
Ans: "char str[] = "Hello"; or char str[10] = "Hello";"

Q41: "Difference between character arrays and strings."
Ans: "String is a char array ending with '\0'; char array may not be null-terminated."

Q42: "Read and print strings in C."
Ans: "Use scanf("%s", str) and printf("%s", str); fgets() for spaces."

Q43: "Concatenate two strings in C."
Ans: "Use strcat(str1, str2) from <string.h>."

Q44: "Program to find length of string without strlen."
Ans: "Iterate with a loop until '\0' to count characters."

Q45: "Array of pointer and pointer of array."
Ans: "Array of pointers: char *arr[5]; Pointer to array: char (*ptr)[5];"

Q46: "Find smallest and largest element in array."
Ans: "Iterate array, keep track of min and max variables."

Q47: "What are tokens in C?"
Ans: "Tokens are smallest units: keywords, identifiers, constants, operators, punctuators."

Q48: "Difference between continue and break statements."
Ans: "continue skips current iteration; break exits the loop."

Q49: "Difference between pre-increment and post-increment."
Ans: "Pre (++i) increments then uses value; Post (i++) uses value then increments."

Q50: "Difference between type casting and type conversion."
Ans: "Type casting is explicit; type conversion can be implicit or explicit."

51. Basic data types supported in C

C supports primary data types: int, char, float, double. Other derived types include arrays, pointers, structures, unions. These can have signed or unsigned variants.

52. Convert a string to numbers in C

Use atoi() for integers, atof() for float/double, or strtol()/strtod() for more control.

Example:

```
char str[] = "123";

int num = atoi(str); // num = 123
```

53. Local vs Global variables

- **Local variable: Declared inside a function; scope limited to that function; stored on the stack.**
- **Global variable: Declared outside all functions; accessible throughout the program; stored in data segment.**

54. typedef in C

typedef allows giving custom names to existing data types.

```
typedef unsigned int uint;

uint x = 100;
```

55. Enumerations

enum defines a set of named integer constants.

```
enum Week {Mon, Tue, Wed};

enum Week today = Tue;
```

56. Infinite loop in C

Can be created using:

```
while(1) { }

for(;;) { }
```

57. getc(), getchar(), getch(), getche()

- **getc() – reads a character from a file.**
- **getchar() – reads a character from stdin.**
- **getch() – reads a character without echo (console).**
- **getche() – reads a character with echo.**

58. Recursion in C

A function calling itself is recursion.

Example: Factorial:

```
int fact(int n){ if(n<=1) return 1; return n*fact(n-1); }
```

59. Preprocessor directives

Instructions processed before compilation, e.g., #include, #define, #ifdef.

60. Compile without main()

Yes, some compilers allow compiling without main() for library functions, but program execution requires main().

61. Static variables

- **Static inside function: Retains value between calls.**
- **Static outside function: Scope limited to file.**

62. malloc() vs calloc()

- **malloc(size) – allocates uninitialized memory.**
- **calloc(n, size) – allocates zero-initialized memory.**

63. Dangling pointers vs memory leaks

- **Dangling pointer: Points to freed memory.**
- **Memory leak: Memory allocated but not freed, losing reference.**

64. Pointer to pointer

A variable that stores the address of another pointer.

```
int x = 10;

int *p = &x;

int **pp = &p;
```

65. Pointer declaration

```
int *p; // pointer to int

char *c; // pointer to char
```

66. Pointer arithmetic

You can perform +, - on pointers considering the size of the data type.

```
p++; // moves pointer to next int (if p is int*)
```

67. Null pointers & void pointers

- **Null pointer: Pointer not pointing anywhere (NULL).**
- **Void pointer: Generic pointer; needs casting before dereferencing.**

68. Passing pointers to functions

Pass address of variable to allow function to modify it.

```
void change(int *p){ *p = 100; }
```

69. NULL pointer

A pointer initialized to NULL represents no memory address.

70. Huge pointers

Used in DOS programming for addressing memory beyond 1MB. Rarely used today.

71. Dynamic data structure

Memory is allocated at runtime, e.g., linked lists, trees.

72. Near, far, huge pointers

Used in 16-bit DOS memory models:

- Near: 16-bit offset
- Far: 32-bit segment:offset
- Huge: normalized far pointer

73. Memory leak & prevention

Occurs when allocated memory isn't freed. Use free() for prevention.

74. Structure & Union

- Structure: Stores multiple data members; each has separate memory.
- Union: Multiple members share same memory.

75. Struct vs Union

- Struct: All members occupy separate memory; size = sum of members.
- Union: All members share memory; size = largest member.

76. r-value & l-value

- l-value: Memory location that can appear on left side of assignment.
- r-value: Value/expression that appears on right side.

77. Basic data types in C

int, char, float, double, void (primary types).

78. Pointer

A variable storing the address of another variable.

79. Reference vs Pointer

- C does not support references (C++ does).
- Pointer: variable holding address, can be reassigned.

80. malloc vs calloc

- malloc(size) → uninitialized memory
- calloc(n, size) → zero-initialized memory

81. Segmentation fault

Occurs when accessing invalid memory, e.g., dereferencing NULL pointer.

82. sizeof operator

Returns memory size (in bytes) of data type or variable.

83. Constant declaration

const int x = 10;

#define PI 3.14

84. Storage classes

auto, register, static, extern — define scope & lifetime of variables.

85. == vs =

- **= : assignment operator**
- **== : equality comparison**

86. Null pointer

Pointer pointing to no memory, initialized as NULL.

87. Array of pointers vs pointer of array

- **Array of pointers: int *arr[10]; → array containing pointers.**
- **Pointer to array: int (*p)[10]; → points to whole array.**

88. Pass pointer to function

void func(int *p){ *p = 5; }

89. Function pointer

Pointer storing address of a function:

int (*fptr)(int, int) = ∑

90. volatile keyword

Tells compiler not to optimize the variable; its value may change externally.

91. Dynamic memory allocation

Use malloc(), calloc(), realloc() to allocate memory at runtime.

92. static keyword

- **Inside function: value persists between calls**
- **Outside function: scope limited to file**

93. Recursion example

Factorial example:

int fact(int n){ if(n<=1) return 1; return n*fact(n-1);}

94. Stack vs heap memory

- **Stack: automatic variables, fast, limited size**
- **Heap: dynamic memory, slower, managed manually**

95. Memory leak & prevention

Occurs when memory isn't freed; prevent by using free() after malloc.

96. Deep copy vs shallow copy

- Shallow copy: copies pointer only, not actual data
- Deep copy: allocates new memory, copies data

97. extern keyword

Declares variable defined in another file.

98. const with pointers

- const int *p → value cannot change
- int * const p → pointer cannot change
- const int * const p → neither value nor pointer changes

99. fgets vs gets

- gets() → unsafe, no buffer limit
- fgets() → safe, buffer size specified

100. Error handling in C

Check return values, use errno, perror(), assert(), or custom error messages.

101. Function prototype declaration

```
int add(int a, int b); // declares function before usage
```

102. Union & usage

Union stores different data types in same memory. Useful when only one value is active at a time.

103. goto statement

Transfers control to a labeled statement. Often discouraged as it reduces readability.

104. String to integer conversion

Use atoi() or strtol(). Example:

```
int num = atoi("123");
```

105. File operations

Use fopen(), fclose(), fread(), fwrite(), fprintf(), fscanf().

106. printf vs sprintf

- printf() → prints to console
- sprintf() → prints to string buffer

107. Static function scope

Function accessible only within the file it's defined.

108. File handling

```
FILE *fp = fopen("file.txt", "r");

fread(...); fwrite(...);

fclose(fp);
```

109. Text vs binary file mode

- **Text mode: line endings translated, ASCII format**
- **Binary mode: raw data, no translation**

110. Error handling in files

Check NULL pointer after fopen(), return values for read/write operations.

111. Array of structures

struct Student s[10]; // array of 10 student structures

112. Pointer array of structures

struct Student *s[10]; // array of 10 pointers to structures

113. Compilation process in C

1. **Preprocessing: Handles #include, #define.**
2. **Compilation: Converts .c code to assembly.**
3. **Assembly: Converts assembly to object code (.obj).**
4. **Linking: Combines object code and libraries to executable.**