C Programming Language — 75 Essential Questions & Answers

Prepared for study & exam revision • Generated on 2025-08-25

Q1. What is the difference between variable and constant?

A variable is a named storage location whose value can change during program execution. A constant is a fixed value that cannot change once defined. Variables are declared with a type (e.g., int x;), while constants can be created using the const qualifier (e.g., const int DAYS = 7;) or with #define macros (preprocessor-time textual substitution).

Use const (preferred) for type-checked, scope-respecting constants; use #define for macro-like symbolic names.

```
int x = 5;      /* variable: value can change */
x = 9;

const float PI = 3.14159f;  /* typed constant (read-only) */
/* PI = 4.2; // error: assignment of read-only variable */

#define MAX 100  /* macro constant: replaced by preprocessor */
```

Q2. Which operators can be used with character data type in C Language?

In C, char is an integer type. All integer operators apply: arithmetic (+, -, *, /, %), unary (++/--), relational (==, !=, <, >, <=, >=), logical (&&, ||, ! with nonzero considered true), bitwise (&, |, ^, ~), and shift (<<, >>). Character literals (e.g., 'A') are of type int.

```
char c = 'A';         /* 65 in ASCII */
c++;                  /* arithmetic */
if (c >= 'A' && c <= 'Z') { /* relational & logical */ }
c = c ^ 32;           /* bitwise: toggle case for ASCII letters */
```

Q3. What are rules for naming variables in C Language?

- Can contain letters (A–Z, a–z), digits (0–9), and underscore (_); must not start with a digit.
- Case-sensitive (total, Total, TOTAL are distinct).
- Cannot be a keyword (e.g., int, return, if).
- No spaces or special characters (e.g., $, -, .).
- Length: modern compilers allow long names; older standards guaranteed at least 31 significant characters for external identifiers.

Q4. What is implicit and explicit type conversion?

Implicit conversion (coercion) happens automatically per the usual arithmetic conversions (e.g., int → double in mixed expressions). Explicit conversion (casting) is specified by the programmer and may change how the bits are interpreted or truncate values.

```
double d = 3;          /* implicit: int 3 -> double 3.0 */
int x = 3.7;           /* implicit: truncates to 3 */

double y = (double)5/2;  /* explicit: 5 becomes 5.0 => result 2.5 */
int   z = (int)3.99;     /* explicit: 3 */
```

Q5. What is hierarchy and associativity in C Language?

Hierarchy (precedence) determines which operators bind first. Associativity resolves ties of same-precedence operators (e.g., left-to-right for +, -, *, /; right-to-left for = and ?:).

Parentheses always override precedence; use them to make intent explicit.

- Highest: () (function call), [] (subscript), . ->, postfix ++/--
- Unary: + - ! ~ (type) sizeof ++/-- (prefix)
- Multiplicative: * / % (left-to-right)
- Additive: + - (left-to-right)
- Shifts: << >> (left-to-right)
- Relational: < <= > >= (left-to-right)
- Equality: == != (left-to-right)
- Bitwise: & then ^ then | (each left-to-right)
- Logical: && then || (left-to-right, short-circuit)
- Conditional ?: (right-to-left)
- Assignment: =, +=, ... (right-to-left)
- Comma operator , (left-to-right)

Q6. What is type promotion and what is type casting?

Type promotion is the automatic conversion to a wider type (e.g., char/short → int) in expressions; this prevents overflow of small types during arithmetic. Type casting is an explicit request to convert a value to another type using (new_type)expr.

```
char c = 120, d = 10;
int s = c + d;   /* c and d promoted to int, s = 130 */

float f = (float) (5/2); /* 5/2 = 2 (int division) -> cast later => 2.0 */
float g = 5.0f/2;        /* promotion to float => 2.5 */
```

Q7. What is keyword in C Language?

A keyword is a reserved word with special meaning in the language and cannot be used as an identifier. Examples: int, float, if, else, for, while, return, switch, case, sizeof, static, extern, const, void, enum, struct, union, typedef, break, continue, goto.

Q8. What will be effect of the following statement “int a = b = c = d = 10”?

Assignment associates right-to-left, so d = 10, then c = (d = 10) → 10, b = 10, and finally a is initialized to 10.

However, in C this statement is valid only if b, c, and d are already declared. The declaration 'int a = b = c = d = 10;' declares only a; using undeclared b/c/d is a compile-time error.

If b, c, and d are previously declared (e.g., as int), then all four become 10.

```
int b, c, d;   /* must exist already */
int a = b = c = d = 10; /* all become 10 */
```

Q9. What is mixed mode arithmetic and how is it solved, explain with example?

Mixed-mode arithmetic involves operands of different types (e.g., int with double). C applies the usual arithmetic conversions to bring operands to a common type (usually the widest).

Integer types are promoted; if any operand is double, the other becomes double; if float and int, int becomes float; etc.

```
int   i = 5;
double d = 2.0;
double r = i / d;  /* i promoted to double => 5.0/2.0 = 2.5 */

int a = 5, b = 2;
double x = (double)a/b; /* 2.5; casting avoids integer division */
```

Q10. What is definition of true and false in C Language?

In C, zero is false and any nonzero value is true. Relational and logical operators yield 0 or 1. With &lt;stdbool.h&gt;, bool is an alias of _Bool and the macros true and false expand to 1 and 0.

```
#include <stdbool.h>
bool ok = (5 > 3);  /* ok == true (1) */
if (ok) { /* ... */ }
```

Q11. What are the logical operators in C explain with an example?

Logical AND (&&), logical OR (||), and logical NOT (!). They short-circuit: evaluation stops as soon as the result is determined.

```
int x = 0, y = 10;
if (x != 0 && (10/y) > 1) { /* left side false, right side NOT evaluated */ }
if (x == 0 || y == 0) { /* true because x==0; y==0 not evaluated */ }
if (!x) { /* !0 -> 1 (true) */ }
```

Q12. What is the conditional operator and when should we use it?

The conditional operator ?: selects between two expressions based on a boolean condition. Use it for concise value selection, not for complex statements (which hurt readability).

```
int a = 5, b = 10;
int max = (a > b) ? a : b;
printf("max=%d\n", max);
```

Q13. Explain working of while and for loop with help of example.

A while loop tests the condition first; executes the body while condition is true. A for loop combines initialization, condition, and iteration expressions in one header.

```
/* while */
int i = 0;
while (i < 3) {
    printf("%d ", i);
    i++;
}

/* for */
for (int j = 0; j < 3; j++) {
    printf("%d ", j);
}
```

Q14. What is difference between while and do-while loop?

- while checks the condition first; body may execute zero times.
- do-while executes body first then checks the condition; body executes at least once.

```
int n = 0;
while (n > 0) { printf("never runs"); }   /* zero times */

do { printf("runs once"); } while (n > 0); /* at least once */
```

Q15. What is the difference between pre and post increment and decrement operator?

++i (pre-increment) increments then yields the new value; i++ (post-increment) yields the old value then increments. Similarly for --. Beware of using multiple increments of the same variable within one expression—this can lead to undefined behavior.

```
int i = 5;
printf("%d %d\n", ++i, i); /* prints 6 6 */
i = 5;
printf("%d %d\n", i++, i); /* prints 5 6 */
```

Q16. Can we do multiple initializations, multiple condition and multiple increment in for loop?

Yes. The initialization and iteration fields accept comma-separated expressions. The middle field is a single expression that controls the loop; you can combine conditions with && or ||. (The comma operator in the middle evaluates left then right, returning the rightmost expression's value.)

```
for (int i = 0, j = 10;  i < j && j < 100;  i++, j += 2) {
    /* ... */
}
```

Q17. What is break and continue in C language?

break exits the nearest enclosing loop or switch immediately. continue skips the rest of the current loop iteration and proceeds to the next iteration (re-evaluating the loop condition).

```
for (int i=0; i<10; i++) {
    if (i == 5) break;     /* exits loop */
    if (i % 2 == 0) continue; /* skip even numbers */
    printf("%d ", i);      /* prints 1 3 */
}
```

Q18. How do we use switch in C language?

switch selects among multiple branches based on an integral expression (int, char, enum). Use case labels with constant expressions and break to avoid fall-through.

```
char c = 'b';
switch (c) {
    case 'a': puts("A"); break;
    case 'b': puts("B"); break;
    default : puts("Other"); break;
}
```

Q19. We can have expression in switch condition but not in switch case, what does it mean?

switch(expression) allows any expression that yields an integral value at runtime. Each case label must be an integer constant expression known at compile time (e.g., literals, enum constants, or macros), not variables or runtime results.

```
int x = 3;
switch (x + 1) {   /* expression OK */
    case 4: break; /* OK: constant literal */
    /* case x: break; // ERROR: not a constant expression */
}
```

Q20. Can we use any type of expression in case of switch, explain with example?

No. Only integer constant expressions are allowed (including char literals and enum constants). Floating-point and non-constant expressions are not allowed. Macros that expand to constants are fine.

```
#define FOUR 4
enum Color { RED=1, GREEN=2 };

int v = 2;
switch (v) {
    case FOUR:     /* OK (macro -> 4) */ break;
    case 'A':      /* OK (char literal) */ break;
    case GREEN:    /* OK (enum constant) */ break;
    /* case v+1:  // ERROR: not constant */
    /* case 3.14: // ERROR: not integer constant */
}
```

Q21. How are break and continue used in switch?

break exits the switch. continue is not meaningful in a stand-alone switch; if used inside a switch within a loop, continue applies to the enclosing loop, not the switch.

```
for (int i=0; i<5; i++) {
    switch (i) {
        case 2: continue; /* skips to next loop iteration */
        case 3: break;    /* exits the switch only */
    }
    printf("%d ", i);     /* prints 0 1 4 */
}
```

Q22. How is goto used in C language, explain with example?

goto transfers control to a labeled statement in the same function. It should be used sparingly (e.g., to break out of deep nesting or for cleanup on error).

```
int rc = -1;
FILE *f = fopen("data.txt", "r");
if (!f) goto done;
/* ... many steps ... */
rc = 0;
done:
if (f) fclose(f);
return rc;
```

Q23. What are pointers in C language?

A pointer holds the memory address of another object or function. Pointers are declared with * and must be dereferenced (*) to access the pointed value.

```
int x = 42;
int *p = &x;    /* p stores address of x */
printf("%d\n", *p); /* dereference: prints 42 */
```

Q24. What operations are allowed on pointer in C language?

- Assignment and copy between compatible pointer types.
- Comparison (==, !=, <, >) for pointers to the same array/object.
- Arithmetic: ++/--, +/– integer, difference between two pointers to elements of the same array (ptrdiff_t).
- Dereference (*) to access the pointed value; member access via -> for pointers to structs.

```
int a[5];
int *p = a;  /* &a[0] */
p++;         /* points to a[1] */
int diff = &a[4] - &a[1]; /* 3 elements apart */
```

Q25. What is function declaration?

A function declaration (prototype) tells the compiler the function’s name, return type, and parameter types, enabling type checking across translation units.

```
/* header.h */
int add(int a, int b);  /* declaration/prototype */
```

Q26. What is function definition and function call?

The function definition provides the body (implementation). A function call transfers control to the function; when it returns, execution resumes.

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

int s = add(2, 3);     /* call */
```

Q27. What are the benefits of using functions in programs?

- Modularity and reusability of code.
- Abstraction and improved readability.
- Easier testing and debugging.
- Encourages separation of interface and implementation.
- Supports recursion and divide-and-conquer patterns.

Q28. How many values can a function return in C explain with example?

A function returns a single value via its return type. To return multiple values, use pointers (output parameters) or return a struct.

```
/* via pointers */
void divrem(int a, int b, int *q, int *r) {
    *q = a / b; *r = a % b;
}

/* via struct */
typedef struct { int q, r; } Pair;
Pair divrem2(int a, int b) { return (Pair){ a/b, a%b }; }
```

Q29. What is a function prototype what is use of same?

A prototype is a declaration with parameter types (and optionally names). It enables compile-time checking of argument count/types and proper default promotions.

It supports separate compilation by placing prototypes in headers for use across files.

Q30. What is call by value and call by reference.

C uses call-by-value: arguments are copied into parameters. To simulate call-by-reference (so the callee can modify the caller’s variables), pass pointers.

```
void set42(int *p) { *p = 42; }
int x = 0;
set42(&x);  /* x becomes 42 */
```

Q31. What do mean by pointer to pointer to integer pointer who will you use it?

A pointer to pointer (e.g., int **pp) stores the address of another pointer (int *p). A 'pointer to pointer to integer pointer' could be interpreted as int ***ppp.

Use cases: functions that need to modify a pointer (e.g., allocate and return a buffer), dynamic 2D arrays (array of pointers), or data structures where you need to update a head pointer.

```
/* function modifying a pointer */
int alloc_array(int **out, size_t n) {
    *out = (int*)malloc(n * sizeof(int));
    return *out ? 0 : -1;
}

/* double-pointer for 2D arrays */
int **rows = malloc(R * sizeof(*rows));
for (size_t i=0;i<R;i++) rows[i] = malloc(C * sizeof(*rows[i]));
```

Q32. What is recursion?

Recursion is a technique where a function calls itself to solve a problem by reducing it to smaller subproblems. It requires a base case to stop and a step that progresses toward it.

```
int fact(int n) {
    if (n <= 1) return 1;      /* base case */
    return n * fact(n-1);      /* recursive step */
}
```

Q33. How is stack used in recursion explain with example?

Each function call creates a stack frame storing parameters, local variables, and return address. Recursive calls push new frames; when a call returns, its frame is popped (LIFO).

```
/* For fact(3): frames push like fact(3)->fact(2)->fact(1), then unwind. */
```

Q34. What are all data types in C language with range of each?

Ranges are implementation-defined. Typical ranges on a 32-bit int / 64-bit long long system (two’s complement):

- char: signed –128..127, unsigned 0..255
- short: signed –32768..32767, unsigned 0..65535
- int: signed –2,147,483,648..2,147,483,647; unsigned 0..4,294,967,295
- long: typically –2,147,483,648..2,147,483,647 on ILP32; on LP64, long is 64-bit
- long long: –9,223,372,036,854,775,808..9,223,372,036,854,775,807; unsigned 0..2^64–1

- float: ~1.2e–38..3.4e38 (≈6–7 decimal digits)
- double: ~2.2e–308..1.8e308 (≈15–16 digits)
- long double: implementation-defined (80-bit/128-bit on some systems)
- _Bool/bool: 0 or 1

Q35. What are storage classes in C language?

Storage classes control scope, linkage, and lifetime: auto (default for block), register (hint to store in CPU register; may be ignored), static, and extern. (C11 also adds _Thread_local).

Q36. What is external storage class, when should we use it?

extern declares a variable or function that is defined in another translation unit or later in the same file. Use extern in headers to share globals across files without defining them multiple times.

```
/* file1.c */ int counter = 0;   /* definition */
/* file2.c */ extern int counter; /* declaration */
```

Q37. What is static storage class, when should we use it?

static on a global gives internal linkage (visible only in that file). static on a local variable gives it static storage duration (retains value across calls) while keeping block scope.

```
/* file scope */
static int module_only = 0; /* not visible from other files */

void f(void) {
    static int calls = 0;  /* preserved between calls */
    calls++;
}
```

Q38. What is scope and life of a variable, explain with example?

Scope is where the name is visible; lifetime is how long the storage exists.

Automatic (block) variables: scope = block; lifetime = during block execution. Static variables: scope per declaration (file/global or block), lifetime = entire program. Globals have external or internal linkage.

```
void g(void) {
    int x = 0;     /* block scope, auto lifetime */
    static int y;  /* block scope, static lifetime (zero-initialized) */
}
```

Q39. What is the default initial values of all storage classes in C language?

Automatic (auto/register) local variables: indeterminate (garbage). Static-duration objects (global/static): zero-initialized by default (ints -> 0, pointers -> NULL, chars -> '\0').

Q40. What is preprocessor in C language, what work does it do?

The preprocessor runs before compilation. It handles macro expansion (#define), file inclusion (#include), conditional compilation (#if/#ifdef), and pragmas (#pragma). It produces a translation unit for the compiler.

Q41. What are macros with arguments, explain with example?

Macros with arguments are preprocessor substitutions that accept parameters. Always parenthesize arguments and the whole expression to avoid precedence bugs.

```
#define SQR(x)   ((x) * (x))
#define MAX(a,b) ((a) > (b) ? (a) : (b))

int a = SQR(1+2);     /* expands to ((1+2)*(1+2)) -> 9 */
int m = MAX(x++, y++);/* beware: x++ may be evaluated twice */
```

Q42. What is conditional compilation in C language?

Conditional compilation includes/excludes code based on compile-time conditions using #if, #ifdef, #ifndef, #elif, #else, #endif.

```
#ifdef DEBUG
  printf("debug: x=%d\n", x);
#endif
#if defined(WIN32) || defined(_WIN32)
  /* Windows-specific code */
#else
  /* POSIX code */
#endif
```

Q43. What is pragma directive in C?

#pragma provides implementation-specific instructions to the compiler (non-portable). Examples include controlling structure packing or disabling warnings. #pragma once (widely supported) prevents multiple header inclusion (non-standard but common).

```
#pragma pack(push, 1)
struct P { char c; int i; };
#pragma pack(pop)

#pragma once  /* in a header file: common extension */
```

Q44. What is difference between #include &lt;stdio.h&gt; and #include "stdio.h"

&lt;...&gt; tells the preprocessor to search system include paths only. "..." searches the current directory first, then system paths. Use &lt;...&gt; for standard headers and "..." for your own headers.

Q45. What are the differences between function and macro, which one should we use in which situation?

- Macro: expanded by preprocessor; no type checking; can evaluate arguments multiple times; no call overhead; can bloat code.
- Function: compiled entity with types; one evaluation of arguments; call/return overhead (often optimized with inline).
- Prefer functions for safety and clarity; use macros for small, header-only, generic operations when necessary—or prefer inline functions in modern C.

Q46. What are arrays, how is memory allocated to an array?

An array is a contiguous sequence of elements of the same type. For static arrays (fixed size), memory is allocated either in static storage (if at file scope or static) or on the stack (if local automatic). Variable Length Arrays (C99) are allocated at runtime on the stack. Dynamic arrays are allocated on the heap via malloc/calloc.

```
int a[5];         /* 5 ints, contiguous */
```

```
static int s[10]; /* static storage duration */
int n = 20; int vla[n]; /* VLA (C99) */
int *p = malloc(n * sizeof *p); /* dynamic */
```

Q47. How are pointers used with arrays?

Array names decay to pointers to their first element in most expressions. Pointer arithmetic moves by element size.

```
int a[3] = {10,20,30};
int *p = a;        /* same as &a[0] */
*(p+1) = 99;       /* sets a[1] */
printf("%d\n", p[2]); /* 30 */
```

Q48. How do you pass a complete array to a function with help of pointer?

Functions cannot take arrays by value; they receive a pointer to the first element along with the length. For 2D arrays, you must specify column size (or pass a pointer to array).

```
void sum1D(const int *a, size_t n) {
    long s=0; for (size_t i=0;i<n;i++) s+=a[i];
    printf("%ld\n", s);
}

/* 2D: known column size */
void print2D(int a[][3], size_t rows) {
    for (size_t i=0;i<rows;i++) for (size_t j=0;j<3;j++) printf("%d ", a[i][j]);
}

/* 2D using pointer to array */
void print2D_p(int (*a)[3], size_t rows) { /* ... */ }
```

Q49. What is bound checking in array, how is it implemented in C language?

Bounds checking means verifying index is within valid range [0, n). C does not enforce bounds checking automatically; the programmer must check indices to avoid undefined behavior and security bugs.

```
if (idx >= 0 && idx < n) a[idx] = value; else /* handle error */;
```

Q50. How is 2-dimensional array initialized in C language, explain with example?

Use nested braces, or a flat list in row-major order. The second dimension must be specified for declarations (except when initializing at definition).

```
int m[2][3] = { {1,2,3}, {4,5,6} };
int n[][3]   = { 1,2,3, 4,5,6 }; /* same */
```

Q51. How pointers are used to access 2-dimentional array?

A 2D array int a[R][C] is an array of R elements, each of which is an array of C ints. A pointer to its rows has type int (*)[C].

```
int a[2][3] = { {1,2,3}, {4,5,6} };
int (*p)[3] = a;       /* pointer to array[3] */
printf("%d\n", p[1][2]); /* 6 */

/* Flattened access: */
int *q = &a[0][0];
printf("%d\n", q[1*3 + 2]); /* row-major index */
```

Q52. What is array of pointers how is it used?

An array of pointers stores addresses of objects/strings. Useful for jagged arrays or lists of strings.

```
const char *months[] = {"Jan","Feb","Mar"}; /* array of 3 pointers to char */
int *rows[10]; /* each rows[i] can point to a dynamically sized row */
```

Q53. What is searching, what are types of searching?

Searching finds the position of a target. Common techniques: linear search (unsorted), binary search (sorted). Hashing provides average O(1) lookup via a hash table.

```
/* linear search */
int find(const int *a, size_t n, int key) {
    for (size_t i=0;i<n;i++) if (a[i]==key) return (int)i;
    return -1;
}
```

Q54. What is sorting, what are types of sorting you know, which sorting algorithm is the most efficient and why?

Sorting rearranges items in order. Algorithms include bubble, selection, insertion, merge, quick, heap, counting, radix, bucket.

There is no single 'most efficient' in all cases. For general comparison sorting, O(n log n) is optimal (merge/heap/quick). Quicksort is fast on average and in-place but has worst-case O(n^2); mergesort is stable with O(n log n) worst-case but needs extra space; heapsort guarantees O(n log n) and is in-place but has larger constants. For integers with bounded range, counting/radix can achieve O(n + k).

Q55. What is definition of string in C language how is it different from array?

A C string is a contiguous array of char terminated by a null byte '\0'. Any char array can hold arbitrary data, but it is a string only if terminated with '\0'.

```
char s[5] = "hi"; /* actually {'h','i','\0','\0','\0'} */
```

Q56. What is difference between string and character pointer used to store string?

char s[] = "hi" creates a modifiable array initialized with a copy of the literal. char *p = "hi" points to a string literal with static storage that must not be modified.

```
char s[] = "hi";  /* OK to modify s[0]='H'; */
char *p  = "hi";  /* UB to modify *p; literals are read-only */
```

Q57. What is strnset() function in C?

strnset(dest, ch, n) (non-standard; available in some compilers like MSVC) sets at most n characters of dest to ch until a '\0' is encountered. It is not part of the ISO C standard. Use memset or a loop instead.

```
/* portable alternative */
void strnset_portable(char *s, int ch, size_t n) {
    for (size_t i=0; i<n && s[i] != '\0'; i++) s[i] = (char)ch;
}
```

Q58. How to use array of pointer to store multiple strings?

Use an array of char* pointing to string literals or dynamically allocated copies; or use a 2D char array for fixed-width storage.

```
/* pointers to string literals */
const char *names[] = {"Alice", "Bob", "Carol"};

/* dynamic copies */
char *names2[3];
/* strdup is POSIX; if unavailable, implement your own */
names2[0] = strdup("Alice"); names2[1] = strdup("Bob"); names2[2] = strdup("Carol");
/* remember to free() later */
```

Q59. What are limitations of array of pointers to strings?

- If pointing to string literals, you cannot modify the strings (read-only).
- Varying string lengths mean total memory is non-contiguous; more allocations and potential fragmentation.
- Need to manage memory (allocate/free) when strings are dynamic.

Q60. How do you initialize and structure in C language?

Initialize with an initializer list or designated initializers (C99+). Unspecified members default to zero.

```
struct Point { int x, y; };
struct Point p1 = {10, 20};
struct Point p2 = {.y = 5, .x = 1}; /* designated */
```

Q61. What is difference between structure and union?

In a struct, all members exist simultaneously with separate storage; size >= sum of members (plus padding). In a union, all members share the same memory; size equals the largest member. Only one member is 'active' at a time.

```
struct S { int i; double d; };   /* size at least sizeof(int)+sizeof(double) */
union U { int i; double d; };    /* size == max(sizeof(int), sizeof(double)) */
```

Q62. What are uses of union in C language?

- Memory-saving variant types (tagged unions).
- Interpreting the same memory in different ways (with care; watch strict aliasing).
- Low-level/embedded programming for registers/bit-fields.

Q63. How do you copy one structure variable to other structure variable?

Structures of the same type can be assigned directly (shallow copy). If the struct contains pointers, consider deep copying the pointed data as needed.

```
struct P { int x; int *ptr; };
struct P a = {1, NULL}, b;
b = a;   /* OK: member-wise copy */
/* Deep copy example would allocate and copy *ptr separately. */
```

Q64. What is static and dynamic memory allocation in C language?

Static allocation occurs at compile/link time (globals, static locals). Automatic (stack) allocation occurs when entering a block (local variables). Dynamic allocation occurs at runtime from the heap using malloc/calloc/realloc/free.

Q65. What are functions used for dynamic memory allocation in C language?

- void *malloc(size_t size) — allocates uninitialized memory.
- void *calloc(size_t nmemb, size_t size) — allocates zero-initialized array.
- void *realloc(void *ptr, size_t size) — resizes allocation.
- void free(void *ptr) — releases memory.

Q66. What is difference between malloc() and calloc() function in C language?

- malloc(size) allocates size bytes; contents are indeterminate.
- calloc(n, size) allocates n*size bytes and initializes all bits to zero (sets integers/pointers to 0, chars to '\0').
- calloc also checks for multiplication overflow in some implementations; always validate the result.

Q67. What is void pointer, how is it used?

A void* is a generic pointer that can hold the address of any object type. You must cast it to the correct type before dereferencing. Pointer arithmetic is not allowed directly on void* (GNU extension allows it, standard C does not).

```
void *vp = malloc(100);
int  *ip = (int*)vp;  /* cast before dereference */
free(vp);
```

Q68. What are the escape sequences in C language?

- \n (newline), \t (tab), \r (carriage return), \b (backspace), \f (form feed), \v (vertical tab), \\ (backslash), \' (single quote), \" (double quote), \? (question mark)
- \0 (null), \ooo (octal byte), \xhh (hex byte)

Q69. What is enumerated data type in C language, when should you use it?

An enum is a user-defined integer type with named constants. Use it to make code more readable and to restrict values to a set of named options.

```
enum State { ST_IDLE, ST_RUN, ST_STOP };
enum State s = ST_IDLE;
```

Q70. Which functions will you use formatted output in C language, explain with example?

Use printf (stdout), fprintf (to FILE*), sprintf/snprintf (to string). Prefer snprintf to avoid buffer overflows. Use correct format specifiers for types.

```
int a = 42;
double d = 3.14;
printf("a=%d d=%.2f\n", a, d);
FILE *f = fopen("out.txt", "w");
```

```
fprintf(f, "Answer: %d\n", a);
char buf[32];
snprintf(buf, sizeof buf, "Pi=%.3f", d);
```

Q71. What are different modes of opening files in C language?

- "r", "w", "a" — read, write (truncate/create), append
- "r+", "w+", "a+" — read/write variants
- Add "b" for binary (e.g., "rb", "wb")

Q72. Why binary files are more efficient than text files?

Binary files store data in its raw binary form without text conversion. This reduces file size for numeric data and avoids parsing/formatting overhead. However, binary files are less portable/human-readable and can be sensitive to endianness and representation.

Q73. How do you use command line arguments in C language?

Use main(int argc, char *argv[]) to receive the argument count and vector. argv[0] is the program name; the rest are arguments.

```
int main(int argc, char *argv[]) {
    for (int i=0; i<argc; i++) printf("arg[%d]=%s\n", i, argv[i]);
    return 0;
}
```

Q74. What are bitwise operators in C language.

- & (AND), **|** (OR), ^ (XOR), ~ (NOT), << (left shift), >> (right shift). Operate on integral types bit-by-bit.

```
unsigned x = 0b0101u, y = 0b0011u;
unsigned a = x & y; /* 0001 */
unsigned o = x | y; /* 0111 */
unsigned e = x ^ y; /* 0110 */
unsigned n = ~x;    /* bitwise NOT */
```

Q75. Explain left shift bitwise operator, what does it do?

x << n shifts the bits of x left by n positions, filling low-order bits with zeros. For unsigned types, this is equivalent to multiplying by 2^n if no overflow occurs. Shifting by a count ≥ the width of the type or shifting a negative value (for signed types) has undefined behavior; signed left shifts that overflow are undefined.

```
unsigned x = 3u;      /* 0000...0011 */
unsigned y = x << 2;  /* 0000...1100 -> 12 */
```