JobDuniya

# Technical Papers C Questions: C Questions Part VI

1. void pascal f (int i, int j, int k) {printf (% d % d % d, i, j, k) ;} void cdecl f (int i, int j, int k) {printf (% d % d % d, i, j, k) ;} main () {int i = 10; f (i ++ , i ++ , i ++) ; printf ( "% d\n" i) ; i = 10; f (i ++ , i ++ , i ++) ; printf ( "% d" i) ;} Answer: 10 11 12 13 12 11 10 13 Explanation: Pascal argument passing mechanism forces the arguments to be called from left to right. Cdecl is the normal C argument passing mechanism where the arguments are passed from right to left.
2. What is the output of the program given below main () {signed char i = 0; for (; i >= 0; i ++) ; printf ( "% d\n" i) ;} Answer 128 Explanation Notice the semicolon at the end of the for loop. THe initial value of the i is set to 0. The inner loop executes to increment the value from 0 to 127 (the positive range of char) and then it rotates to the negative value of-128. The condition in the for loop fails and so comes out of the for loop. It prints the current value of i that is-128.
3. main () {unsigned char i = 0; for (; i >= 0; i ++) ; printf ( "% d\n" i) ;} Answer infinite loop Explanation The difference between the previous question and this one is that the char is declared to be unsigned. So the i ++ can never yield negative value and i >= 0 never becomes false so that it can come out of the for loop.
4. main () {char i = 0; for (; i >= 0; i ++) ; printf ( "% d\n" i) ;} Answer: Behavior is implementation dependent. Explanation: The detail if the char is signed/unsigned by default is implementation dependent. If the implementation treats the char to be signed by default the program will print 128 and terminate. On the other hand if it considers char to be unsigned by default, it goes to infinite loop. Rule: You can write programs that have implementation dependent behavior. But dont write programs that depend on such behavior.
5. Is the following statement a declaration/definition. Find what does it mean? int (∗ x) [10] Answer Definition. x is a pointer to array of (size 10) integers. Apply clock-wise rule to find the meaning of this definition.
6. What is the output for the program given below typedef enum errorType {warning, error, exception,} error; main () {error g1; g1 = 1; printf ( "% d" g1) ;} Answer Compiler error: Multiple declaration for error Explanation The name error is used in the two meanings. One means that it is a enumerator constant with value 1. The another use is that it is a type name (due to typedef) for enum errorType. Given a situation the compiler cannot distinguish the meaning of error to know in what sense the error is used: Error g1; g1 = error; //which error it refers in each case? When the compiler can

distinguish between usages then it will not issue error (in pure technical terms, names can only be overloaded in different namespaces) . Note: The extra comma in the declaration, enum errorType {warning, error, exception,} is not an error. An extra comma is valid and is provided just for programmer's convenience.

7. typedef struct error {int warning, error, exception;} error; main () {error g1; g1. Error = 1; printf ( "% d" g1. Error) ;} Answer 1 Explanation The three usages of name errors can be distinguishable by the compiler at any instance, so valid (they are in different namespaces) . Typedef struct error {int warning, error, exception;} error; This error can be used only by preceding the error by struct kayword as in: Struct error someError; typedef struct error {int warning, error, exception;} error; This can be used only after (dot) . or → (arrow) operator preceded by the variable name as in: g1. Error = 1; printf ( "% d" g1. Error) ; typedef struct error {int warning, error, exception;} error; This can be used to define variables without using the preceding struct keyword as in: Error g1; Since the compiler can perfectly distinguish between these three usages, it is perfectly legal and valid. Note This code is given here to just explain the concept behind. In real programming don't use such overloading of names. It reduces the readability of the code. Possible doesn't mean that we should use it!
8. #ifdef something int some = 0; #endif main () {int thing = 0; printf ( "% d % d\n" some, thing) ;} Answer: Compiler error: Undefined symbol some Explanation: This is a very simple example for conditional compilation. The name something is not already known to the compiler making the declaration int some = 0; effectively removed from the source code.
9. #if something == 0 int some = 0; #endif main () {int thing = 0; printf ( "% d % d\n" some, thing) ;} Answer 0 0 Explanation This code is to show that preprocessor expressions are not the same as the ordinary expressions. If a name is not known the preprocessor treats it to be equal to zero.
10. What is the output for the following program main () {int arr2D [3] [3] printf ( "% d\n" ( (arr2D == * arr2D) , && (* arr2D == arr2D [0] ) ) ) ;} Answer 1 Explanation This is due to the close relation between the arrays and pointers. N dimensional arrays are made up of (N – 1) dimensional arrays. Arr2D is made up of a 3 single arrays that contains 3 integers each. The name arr2D refers to the beginning of all the 3 arrays. * arr2D refers to the start of the first 1D array (of 3 integers) that is the same address as arr2D. So the expression (arr2D == * arr2D) is true (1) . Similarly, * arr2D is nothing but * (arr2D + 0) , adding a zero doesn't change the value/meaning. Again arr2D [0] is the another way of telling * (arr2D + 0) . So the expression (* (arr2D + 0) == arr2D [0] ) is true (1) . Since both parts of the expression evaluates to true the result is true (1) and the same is printed.
11. void main () {if (~0 == (unsigned int) -1) printf (You can answer this if you know how values are represented in memory) ;} Answer You can answer this if you know how values are represented in memory Explanation ~ (tilde operator or bit-wise negation

operator) operates on 0 to produce all ones to fill the space for an integer. 1 is represented in unsigned value as all 1's and so both are equal.

12. int swap (int * a, int * b) {* a =* a +* b; * b =* a-* b; * a =* a-* b;} main () {int x = 10, y = 20; swap (&x, &y) ; printf ( "x =% d y =% d\n" x, y) ;} Answer x = 20 y = 10 Explanation This is one way of swapping two values. Simple checking will help understand this.
13. main () {char * p = ayqm; printf (% c, ++ * (p ++) ) ;} Answer: b
14. main () {int i = 5; printf ( "% d" ++ i ++) ;} Answer: Compiler error: Lvalue required in function main Explanation:+ + i yields an rvalue. For postfix ++ to operate an lvalue is required.
15. main () {char * p = ayqm; char c; c =+ +* p ++ ; printf (% c, c) ;} Answer: b Explanation: There is no difference between the expression ++ * (p ++) and ++ * p ++ . Parenthesis just works as a visual clue for the reader to see which expression is first evaluated.
16. int aaa () {printf (Hi) ;} int bbb () {printf (hello) ;} iny ccc () {printf (bye) ;} main () {int (* ptr [3] ) () ; ptr [0] = aaa; ptr [1] = bbb; ptr [2] = ccc; ptr [2] () ;} Answer: Bye Explanation: Int (* ptr [3] ) () says that ptr is an array of pointers to functions that takes no arguments and returns the type int. By the assignment ptr [0] = aaa; it means that the first function pointer in the array is initialized with the address of the function aaa. Similarly, the other two array elements also get initialized with the addresses of the functions bbb and ccc. Since ptr [2] contains the address of the function ccc, the call to the function ptr [2] () is same as calling ccc () . So it results in printing "bye"
17. main () {int i = 5; printf (% d, i =+ + i == 6) ;} Answer: 1 Explanation: The expression can be treated as i = (++ i == 6) , because == is of higher precedence than = operator. In the inner expression, ++ i is equal to 6 yielding true (1) . Hence the result.
18. main () {char p [] = "% d\n" p [1] = 'c' printf (p, 65) ;} Answer: A Explanation: Due to the assignment p [1] = c 'the string becomes, % c\n. Since this string becomes the format string for printf and ASCII value of 65 is A' the same gets printed.
19. void (* abc (int, void (* def) () ) ) () ; Answer: Abc is a ptr to a function which takes 2 parameters (a) . An integer variable (b) . a ptrto a funtion which returns void. The return type of the function is void. Explanation: Apply the clock-wise rule to find the result.
20. main () {while (strcmp (some, some\0) ) printf (Strings are not equal\n) ;}

Answer: No output

Explanation: Ending the string constant with \0 explicitly makes no difference. So some and some\0 are equivalent. So, strcmp returns 0 (false) hence breaking out of the while loop.