

Q1 The auto keyword can be used in place of 'int', 'double' or other data types to automatically determine the data type of the variable.  
[Automatic type deduction].

Q2  $\text{auto } a = *y$  and  $\text{auto } \&b = *y$   
will return errors. Since  $y$  is of type `int`, and  $*$  is used to return value of an address.  $y$  is not an address  $\rightarrow$  Error.  
 $\text{auto } b = \&y$ . Here  $b$  has type  $\rightarrow \text{int}^*$ .  
 $\&$  is a pointer to an integer.

Q3 Size of struct is sum of sizes of its members, but sometimes larger because of padding.

(a) `double`  $\rightarrow$  8 bytes

`int`  $\rightarrow$  4

`char`  $\rightarrow$  1

`char*`  $\rightarrow$  4 [Address]

17

With padding, it will be greater than this, a multiple of 8.  $\rightarrow$  24.

(b) `int`  $\rightarrow$  4

`double*`  $\rightarrow$  4 [Address]

`char*`  $\rightarrow$  4

`char`  $\rightarrow$  1

13

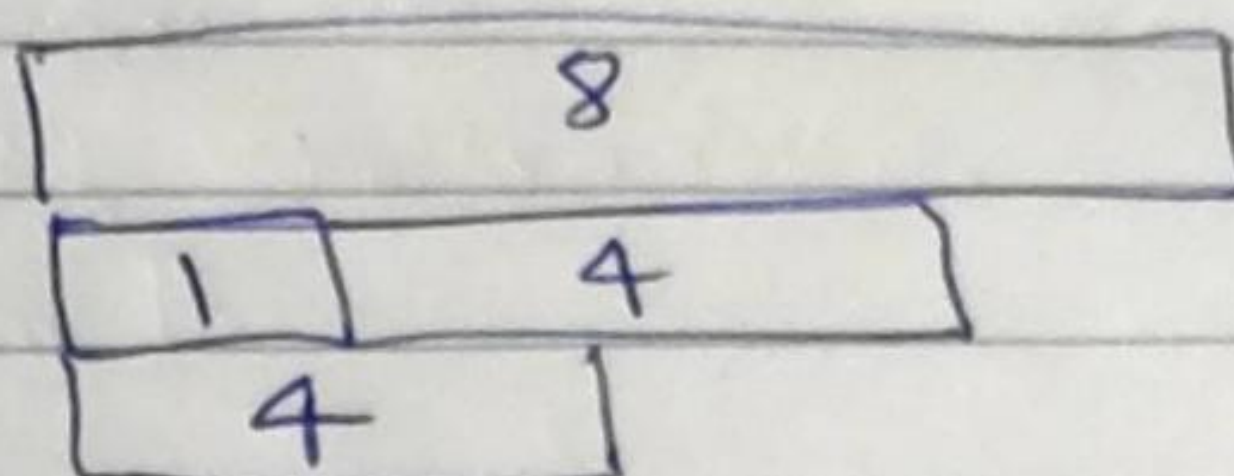
Ans  $\rightarrow$  16

[Multiple of highest individual size].

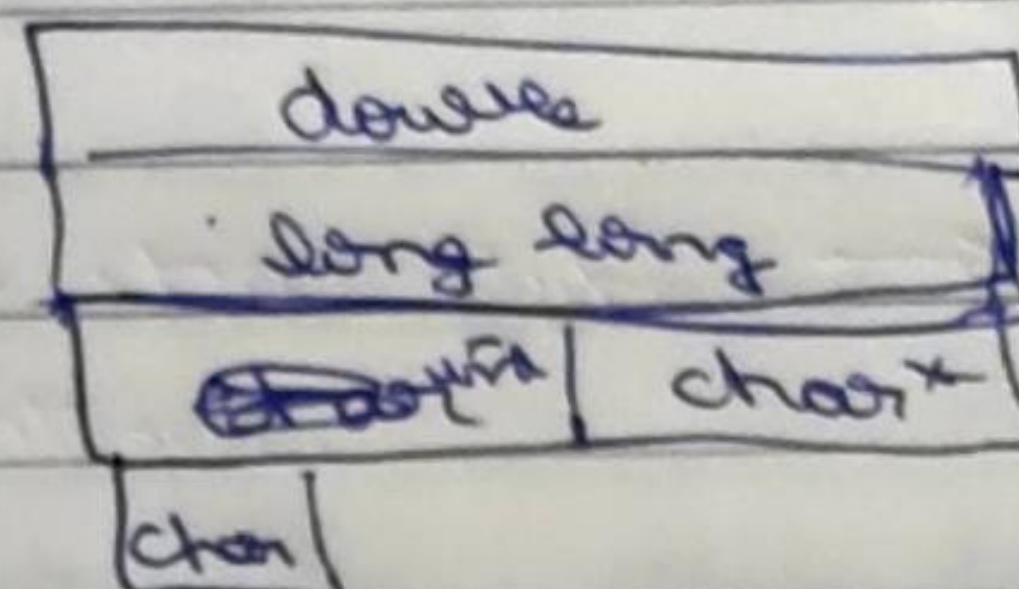


(c) double  $\rightarrow 8$   
 char  $\rightarrow 1$   
 int  $\rightarrow 4$   
 int\*  $\rightarrow 4$   
17

Ans.  $\rightarrow$  24 bytes.



Q4 double  $\rightarrow 8$   
 char  $\rightarrow 1$   
 int  $\rightarrow 4$   
 long long  $\rightarrow 8$   
~~int\*~~ char\*  $\rightarrow 4$



Min. size  $\rightarrow$  32 bytes.

{ double;  
 long long;  
 int;  
 char\*;  
 char;

(a)

Q5 Due to PADDING.



Q6 (a) `int *p = new int;`

Memory is dynamically allocated for pointer `p` but never deleted. ~~So after program~~  
So after the scope of `p` ends, the memory address is lost. This is a memory leak.

(c) ~~(b)~~ `int *p = nullptr;`

This pointer has address 0.

`cout << p;`  $\rightarrow$  Output 0.

A pointer at address 0 is a null pointer.

(b) ~~(a)~~ A dynamically allocated pointer, once deleted, becomes a dangling pointer.

`int *p = new int;`

`delete p;`  $\rightarrow$  // It is now dangling pointer.

Q7 ~~(a)~~ Global scope  $\rightarrow$  ①, ④

Function scope  $\rightarrow$  ②

Local scope  $\rightarrow$  ③, ⑤, ⑥, ⑦, ⑧

All variables inside curly braces have local scope.



Q8 'const'. Non-changing values are best defined as global variables.

Q9 Multiple values can be returned in a ~~to~~ ~~by~~ tuple.

~~by~~ tuple < int, float, string > func ()  
{

return make\_tuple(1, 2.3, "lengthy paper");  
}

Q10 Initialization → Can be used to define a variable and give it a value in one step.

Assignment → Assigns values to a variable that has already been defined.

We can initialize a particular variable only once, but can assign it values multiple times.

```
int x = 5;    // Init.  
x = 109;     // Ass.
```

```
int arr[3] = {4, 2, 0}; // Init  
arr[0] = 92; // Assign.
```



Q11    `int main() {  
    int x = 3;  
    {  
        int x = 5;  
        cout << x; }  
    cout << x;  
    cout << x; }`

Output  $\rightarrow$  53

The outer `x` was hidden ~~at~~ when another `x` was defined.

Global variables can be accessed ~~by~~ outside the scope of the local variable that caused hiding. For example by using a function outside local scope.

Q12 (a) and (b) should give stack overflow as more memory is needed than is available.

(c) will not give stack overflow. The `NEW` keyword lets us dynamically allocate memory from outside the stack.



Q13 delete is used to remove a dynamically allocated pointer.

delete [] is used to delete the entire array.

Using 'delete' on an array can cause memory leak as it will only delete the pointer to the first value in the array and leave the rest.

Q16 Default arguments can be passed in function declaration with default values provided. This default value is used if a value is not given when function is called.

Rule → Default arguments can be set only once, either during declaration or definition.

Q17 Many standard elements of code such as cout, cin, vector, make\_tuple become easier to write if we enter the 'using namespace std;' line.



Q18 ~~2 integers~~  $x$  is initialised twice, and the parameter  $n$  is also initialised twice.  
→ 4 integers copies are made.

~~Q19~~ ~~Output~~ ~~30~~ Q19 Output → 30

Q20  $arr+1$  → gives pointer to the element at index 1.

Precedence of  $&arr$  in  $arr+1$ .  
i.e. →  $(\&arr) + 1$  also points ~~to~~.

$(\&arr) + 1$  points to the end of the array.  
~~Here~~ ~~\* $(\&arr) + 1$~~  ~~- 1~~  
\*  $(ptr-1)$  will point to the last element

Output → 2 5.



Q21 Output  $\rightarrow$   $\begin{matrix} 6 \\ 5 \end{matrix}$  NOT a garbage value.

- ① In function, a copy of the array is passed.  
\*  $\text{arr} = \text{arr} + 1$  means the ~~arr~~ pointer now points to ~~two~~ <sup>the</sup> index [1].
- ② However, outside the function, ~~arr~~ still points to index 0.

Q22

|      |          |   |
|------|----------|---|
| [5]  | delete[] | 4 |
| {15} | delete   | 4 |
| {5}  | delete   | 4 |

Q23 The code decreases the value of  $n$  by a certain amount each step.

~~Answer~~ Let's say  $n$ , in binary, is 10111000.

After first loop,  $n$  becomes 10110000  
second  $\rightarrow$  10100000  
third  $\rightarrow$  10000000  
fourth  $\rightarrow$  00000000

Hence, each time it removes the last set bit.



$$2^3 - 1 \rightarrow 7, 111$$

\* Max possible value for count is :-

Since max value of unsigned int is  $2^{32} - 1$ ,  
which is 1111... (32 times).

Count can be 32.

If count =  $\alpha$ , the operator  $\alpha$  is called a times.

Q24 int delta (char\* c, int size) {  
     wchar\_t <26> b;  
     for (int i = 0; i < size; i++) {  
         b.set (\* (c+i) - 'a');  
     }  
     // ~~cout << b;~~ ~~return~~

// Now b has unique representation. We will do

// binary to decimal conversion.

```

    int ans = 0;
    int m = 1;
    for (int i = 0; i < 26; i++) {
        if (b[i] == 1) ans += m;
        m = m * 2;
    }
    return ans;
}
```