Cpro (Monsoon '24) MidSem: Question & Answers

Instructors: Abhishek Deshpande, Sandeep Nagar, Girish Varma • Course Code: CS0.101 Duration: 60 mins.

Instructions: Before starting writing, fill in your Name, Roll Number, and Seat Location correctly. Keep your exam copy on your desk at all times. Do not remove any pages from this exam booklet.

Student Name:	Roll Number:	Seat Location:

For Evaluators only

No.	1	2	3	4	5	6	7	8	9	10	11	12	Evaluator
MCQ													
SAQ													

Total: / 50

Question 1: (5 marks) Match the following with respect to the following program segment:
int x[3][5] = {{1,2,3,4,5}, {6,7,8,9,10}, {11,12,13,14,15}}, *n=&x;

1. *(*x+2)+5
2. *(*(x)+2) +1
3. *(*(x+1) +3)
4. *n
5. *(n+3) +1

(i) 9
(ii) 13
(iii) 4
(iv) 3
(v) 2
(vi) 12
(vii) 14
(viii) 7
(ix) 1
(x) 8
(xi) 5
(xii) 10
(xiii) 6

Solution: 1. (x) 2. (iii) 3. (i) 4. (ix) 5. (xi)

- $*(*x+2)+5$, This is equivalent to $*(x[0]+2)+5 = *(\&x[0][2]) + 5 = x[0][2] + 5 = 3 + 5 = 8$.
- $((x)+2)+1$, This is equivalent to $*(x[0]+2)+1 = x[0][2] + 1 = 3 + 1 = 4$.
- $((x + 1) + 3)$, This is equivalent to $*(x[1] + 3) = x[1][3] = 9$.
- $*n$, n points to the first element of the 2D array, so $*n = x[0][0] = 1$.
- $*(n+3)+1$, $n+3$ points to the fourth element in the flattened array, which is x[0][3]. So $*(n + 3) + 1 = x[0][3] + 1 = 4 + 1 = 5$

Question 2: (2 marks) Consider that the following statements are used in a program.

(i) sizeof(int);

(ii) sizeof(int*);

(iii) sizeof(int**);

Assuming size of the pointer is 4 bytes and size of int is also 4 bytes, which of the following is true?

(A) Only (i) would compile successfully and it would return size as 4.

(B) (i), ii) and iii) would compile successfully and size of each would be same i.e. 4

(C) (i), ii) and iii) would compile successfully but the size of each would be different and would be decided at run time.

(D) (ii) and iii) would result in a compile error but i) would compile and result in size as 4.

Solution: B.

- $sizeof(int)$: This is valid and will return the size of an integer, which is given as 4 bytes.
- $sizeof(int*)$: This is valid and will return the size of a pointer to an integer. We're told that the size of a pointer is 4 bytes.
- $sizeof(int**)$: This is also valid and will return the size of a pointer to a pointer to an integer. Again, this is a pointer, so it's 4 bytes.

Question 3: (2 marks) What is the return type of the function with prototype: "int func(char x, float v, double t)"?

(A) char

(B) int

(C) float

(D) double

Solution: B.
In C , the return type of a function is specified at the beginning of the function declaration or prototype. The return type is independent of the types of the parameters. Even though the function takes char, float, and double as parameters, it is declared to return an int. Therefore, regardless of what the function does internally or what types of parameters it accepts, this function is defined to return an integer (int) value.

Question 4: (2 marks) Which of the following is a valid function call (assuming the function exists)?

(A) funct;

(B) funct x, y;

(C) funct();

(D) int funct();

Solution: C.

- (A) funct; This is not a valid function call. It's just referencing the function name without calling it.
- (B) funct x, y; This is not a valid function call. It looks like an attempt to declare variables or pass arguments, but it's not using proper function call syntax.
- (C) funct(); This is a valid function call. It calls a function named "funct" with no arguments.
- (D) int funct(); This is not a function call, but a function declaration. It declares a function named "funct" that returns an int and takes no arguments.

Question 5: (2 marks) What is the output of the below code snippet?

```
1 #include <stdio.h>
2
3 void main()
4 {
5   for()
6   printf("Hello");
7 }
```

(A) Infinite loop

(B) Prints "Hello" once.

(C) No output

(D) Compile error

Solution: D.
This code will not compile successfully. The reason is the empty for loop on line 5. In C, a for loop requires a specific syntax: for (initialization; condition; update).

Question 6: (2 marks) What is the return type of *malloc()* ?

(A) void *

(B) Pointer of allocated memory type

(C) void **

(D) int *

Solution: A.
The *malloc()* function in C is used for dynamic memory allocation. It allocates a block of memory of the specified size and returns a pointer to the beginning of that block. The return type of *malloc()* is *void **. This is a pointer to void, which is a generic pointer type in C.
The reason *malloc()* returns *void ** is for flexibility. It can allocate memory for any data type, and the *void ** can be cast to the appropriate pointer type by the programmer.

Question 7: (2 marks) What will be the output produced by the following C code:

```c
int main()
{
int array[5][5];
printf("%d", ((array == *array)
         && (*array == array[0])));
return 0;
}
```

(A) 1

(B) 0

(C) 2

(D) -1

Solution: A.
This code demonstrates a subtle point about array decay and pointer arithmetic in C, showing that for a 2D array, the array name, dereferenced array name, and the first row all point to the same memory location.

- We have a 2D integer array array[5][5]. The *printf* statement is comparing three expressions: array, *array , array[0].
- In C, when used in expressions, an array name decays to a pointer to its first element.
- For a 2D array: array is of type *int (*)[5]* (pointer to an array of 5 ints) **array* is of type *int ** (pointer to int) array[0] is also of type *int ** (pointer to int).
- The key point is that array and **array* are actually the same address - the address of the first element of the 2D array. Similarly, *array[0]* also points to the same address.
- The expression $((array == *array) \&\& (*array == array[0]))$ is comparing these addresses.
- Since all these expressions point to the same memory location, both comparisons will be true. In C, true is represented by 1 and false by 0. The && (logical AND) of two true values is also true.
- Therefore, the expression will evaluate to true, and the output will be 1.

Question 8: (2 marks) What is the output of the following program?

```c
#include <stdio.h>
void func( int *a, int *b)
{
    a = b;
   *a = 2;
}
int main()
{
    int i = 0, j = 1;
    func(&i, &j);
    printf("%d %d", i, j);
    return 0;
}
```

(A) 2 2

(B) 2 1

(C) 0 1

(D) 0 2

Solution: D.
This question tests understanding of pointers and how they behave when passed to functions. It's important to note that changing where a pointer points inside a function doesn't affect the original variable, but changing the value at the address a pointer points to does affect the original variable.

Question 9: (2 marks) What happens when one assigns a value to an element of an array whose subscript exceeds the size of the array?

(A) Nothing, it is done all the time.

(B) Compiler error.

(C) Other data may be overwritten.

(D) The element is set to zero.

Solutions: C.
In C, array bounds are not automatically checked at runtime. Always ensure that your array accesses are within the bounds of the array to write safe and correct code. When you access an array element beyond its declared size (this is called array bounds overflow or buffer overflow), the behavior is undefined according to the C standard.

In practice, what typically happens is: a) The program will access memory that doesn't belong to the array. b) This memory might be used by other variables or data structures in your program. c) Writing to this location will overwrite whatever data was there previously. This can lead to various problems: Corrupted data in other variables, Unexpected program behavior, Program crashes, Security vulnerabilities (in extreme cases). Always ensure that your array accesses are within the bounds of the array to write safe and correct code.

Question 10: (2 marks) What is the output of the following program?

```
int main()
{
    printf("%d %d", (023), (23));
    return 0;
}
```

(A) 023 23

(B) 23 23

(C) 19 23

(D) (023) (23)

Solution: C.

In the given C program, the expression (023) is an octal (base-8) representation, and 23 is a decimal (base-10) number. Octal 023 equals $2 * 8 + 3 = 16 + 3 = 19$ in decimal.

Question-11: (3 marks) What does the following function do?

```
int fun(int x, int y)
{
    if (y == 0)
        return 0;

    return (x + fun(x, y-1));
}
```

(A) $x + y$

(B) $x + x * y$

(C) $x * y$

(D) x^y

Solution: C.

- Base case: If y is 0, the function returns 0. This is the base case that terminates the recursion.
- Recursive case: If y is not 0, the function recursively calls itself with x and $y - 1$. The result of this recursive call is then added to x and returned.

Question-12: (3 marks) What does the following function do?

```
int fun(int n) {
    if (n == 0 || n == 1)
        return n;

    if(n%3 != 0)
        return 0;

    return fun(n/3);
}
```

(A) It returns 1 when n is a multiple of 3, otherwise returns 0.

(B) It returns 1 when n is a power of 3, otherwise returns 0.

(C) It returns 0 when n is a multiple of 3, otherwise returns 1.

(D) It returns 0 when n is a power of 3, otherwise returns 1.

Solution: B.

- Base case: If $n == 0$, it returns 0. If $n == 1$, it returns 1.
- Check if n is divisible by 3: If $n\%3 != 0$ (i.e., n is not divisible by 3), it returns 0.
- Recursive Case: If n is divisible by 3, the function calls itself with $n/3$.

The function checks whether the given number n is a power of 3. If n is a power of 3, it will eventually reduce n to 1 and return 1. If n is not a power of 3, it will return 0 when it finds a number that is not divisible by 3.

Short answer type questions

Question 1 (5 marks)

Explain the difference between call by value and call by reference with an example.
Solution: *Call by Value*: A copy of the argument's value is passed to the function. Any modifications made to the argument within the function do not affect the original value outside the function.
Call by Reference: The address of the argument is passed to the function. Any modifications made to the argument within the function directly affect the original value outside the function. *Example:*

```
#include <stdio.h>
void swapByValue(int x, int y) {
    int temp = x;
    x = y;
    y = temp;
    printf("Inside swapByValue: x = %d, y = %d\n", x, y);
}
void swapByReference(int *x, int *y) {
    int temp = *x;
    *x = *y;
    *y = temp;
    printf("Inside swapByReference: x = %d, y = %d\n", *x, *y);
}
int main() {
    int a = 10, b = 20;

    printf("Before swapByValue: a = %d, b = %d\n", a, b);
    swapByValue(a, b);
    printf("After swapByValue: a = %d, b = %d\n", a, b);

    printf("\nBefore swapByReference: a = %d, b = %d\n", a, b);
    swapByReference(&a, &b);
    printf("After swapByReference: a = %d, b = %d\n", a, b);

    return 0;
}
```

Output:
Before swapByValue: $a = 10, b = 20$
Inside swapByValue: $x = 20, y = 10$
After swapByValue: $a = 10, b = 20$

Before swapByReference: $a = 10, b = 20$
Inside swapByReference: $x = 20, y = 10$
After swapByReference: $a = 20, b = 10$
Note: Call by value is often used when you want to avoid unintended modifications to original variables. Call by reference is useful when you need to modify original variables within a function. Choose the appropriate method based on your specific requirements.

Question 2 (5 marks)

Given an integer array of n elements, write a program to sort the array using insertion sort.

Reference C program:

```
#include <stdio.h>

void insertionSort(int arr[], int n) {
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;
        /* Move elements of arr[0..i-1], that are greater than key,
           to one position ahead of their current position */
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
void printArray(int arr[], int n) {
    int i;
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}
int main() {
    int n, i;
    printf("Enter the number of elements: ");
    scanf("%d", &n);

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

    printf("Original array: ");
    printArray(arr, n);

    insertionSort(arr, n);

    printf("Sorted array: ");
    printArray(arr, n);

    return 0;
}
```

Note: This implementation allows for dynamic input of the array size and elements, making it flexible for various use cases.

Question 3 (5 marks)

Bob has an array consisting of positive and negative numbers. Write a program (without using nested for loops or sorting algorithms or an extra array) to reorder the array elements so that all negative numbers appear before all positive numbers. (Note that the order of elements in the output array is not important)

Reference C program:

```c
#include <stdio.h>
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
void reorderArray(int arr[], int n) {
    int left = 0;
    int right = n - 1;
    while (left < right) {
        // Move left pointer right until we find a positive number
        while (left < n && arr[left] < 0) {
            left++;
        }
        // Move right pointer left until we find a negative number
        while (right >= 0 && arr[right] >= 0) {
            right--;
        }
        // If left is less than right, swap the elements
        if (left < right) {
            swap(&arr[left], &arr[right]);
        }
    }
}
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
int main() {
    int n;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    int arr[n];
    printf("Enter %d integers (positive and negative):\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    printf("Original array: ");
    printArray(arr, n);
    reorderArray(arr, n);
    printf("Reordered array: ");
    printArray(arr, n);
    return 0;
}
```

Question 4 (6 marks)

What is the output returned by $fun(3, 4, 5)$? Answer with justification.

```
#include<stdio.h>
void fun(int n,int a,int b) {
    if (n <= 0)
        return;
    fun(n-1, a, b+n);  // First recursive call
    printf("%d, %d, %d \n",n,a,b); // Print statement
    fun(n-1, b, a+n);  // Second recursive call
}
```

Hints/Step for refrence:
What the function does: It takes three integer parameters: n, a, and b If $n <= 0$, it returns without doing anything. Otherwise, it makes two recursive calls and prints the values of n, a, and b between these calls.
The output will be: (Each line represents a call to *printf()* in the order they occur during the execution of the function).

```
1, 4, 10
2, 4, 8
1, 8, 6
3, 4, 5
1, 5, 9
2, 5, 7
1, 7, 7
```

Follow the recursive calls carefully, noting that the *printf()* statement is executed between the two recursive calls in each non-base case invocation of the function.