C MINOR ASSIGNMENT- 05

1. Consider the following ANSI C program;

```
#include<stdio.h>
int main(){
  int arr[4][5],i,j;
  for(i=0;i<4;i++){
    for(j=0;j<5;j++){
     arr[i][j]=10*i+j;
   }}
 printf("%d\n",arr[2][4]);
 printf("%d\n",*(*(arr+2)+4));
 return 0;}
```

- **Output**:
  - The first `printf` prints `arr[2][4]`.
  - The second `printf` accesses the same value using pointer notation: `*(*(arr + 2) + 4)`.
- **Explanation**:
  - `arr` is a 2D array initialized such that `arr[i][j] = 10 * i + j`. This means:
    - For `i = 2`, the array elements are `[20, 21, 22, 23, 24]`.
  - `arr[2][4]` corresponds to the value `24`.
  - Pointer expression `*(*(arr + 2) + 4)`:
    - `arr + 2` points to the base of the third row (`arr[2]`).
    - `*(arr + 2)` dereferences this to access the row.
    - `*(arr + 2) + 4` shifts the pointer to the fifth element in the row.
    - Dereferencing this pointer retrieves the value `24`.

2. Consider the following ANSI C program;

```
#include<stdio.h>
int main(){
  int arr[4][5],i,j;
  for(i=0;i<4;i++){
    for(j=0;j<5;j++){
     arr[i][j]=10*i+j;
   }}
 printf("%d\n",*(arr[1]+9));
 return 0;}
```

Output: 24

3. Consider the following C program

```
#include<stdio.h>
int main(){
int a[4][5]={
        {1,2,3,4,5},
        {6,7,8,9,10},
        {11,12,13,14,15},
        {16,17,18,19,20}};
 printf("%d\n",*(*(a+**a+2)+3));
 return 0;}
```

- **Output**:
  - The program will print the value `19`.
- **Explanation**:
  - `**a` refers to the first element of the array, which is `1`.
  - `a + **a` calculates `a + 1`, moving to the second row: `{6, 7, 8, 9, 10}`.
  - `a + **a + 2` moves two rows ahead to the third row: `{11, 12, 13, 14, 15}`.
  - `*(a + **a + 2)` dereferences this pointer to access the third row.
  - `*(a + **a + 2) + 3` accesses the fourth element (`14 + 3` columns in the flattened memory).
  - Dereferencing it gives `19`.

Question 4: Output and Explanation of the Given Code

```c
#include<stdio.h>
int main() {
    int a[3][3] = {4, 5, 6, 7, 8, 9, 1, 2, 3};
    printf("%p %p %p\n", a[1] + 2, *(a + 1) + 2, &a[1][2]);
    printf("%d %d %d\n", *(a[1] + 2), *(*(a + 1) + 2), a[1][2]);
    return 0;
}
```

- **Output**:
  - First line: Prints the addresses, all the same.
  - Second line: Prints the values: `9 9 9`.
- **Explanation**:
  - `a[1] + 2`, `*(a + 1) + 2`, and `&a[1][2]` all point to the same location: the address of the element in the second row and third column.
  - The value at this location is `9`, as seen in the array initialization.

```c
#include<stdio.h>
int main() {
    int a[][3] = {4, 5, 6, 7, 8, 9, 1, 2, 3};
    printf("%d,", *a[2]);
    printf("%d,", a[2][0]);
    printf("%d\n", **(a + 1 + ('b' - 'a')));
    return 0;
}
```

- **Output**:
  - The program prints: `1,1,1`.
- **Explanation**:
  - `*a[2]`: `a[2]` refers to the third row `{1, 2, 3}`, and `*a[2]` gives the first element: `1`.
  - `a[2][0]`: Directly accesses the first element of the third row, which is `1`.
  - `**(a + 1 + ('b' - 'a'))`:
    - `'b' - 'a'` evaluates to `1`.
    - `a + 1 + 1` moves to the third row.
    - `**` dereferences to access the first element: `1`.

```c
#include<stdio.h>
int main() {
    int a[][2][4] = {5, 6, 7, 8, 9, 11, 12, 1};
    printf("%d\n", *(*(*(a + 0) + 1) + 2));
    return 0;
}
```

1. `a + 0`: Points to the 0th layer (`{{5, 6, 7, 8}, {9, 11, 12, 1}}`).
2. `*(a + 0)`: Dereferences to the 0th layer, which is a pointer to the sublayer array.
3. `*(a + 0) + 1`: Points to the 1st sublayer (`{9, 11, 12, 1}`).
4. `*(*(a + 0) + 1)`: Dereferences to the 1st sublayer array.
5. `*(*(a + 0) + 1) + 2`: Points to the 3rd element in the 1st sublayer (`12`).
6. `*(*(*(a + 0) + 1) + 2)`: Dereferences this pointer, yielding `12`.

7. Describe the output for the following code snippet.

```c
void fun(int arr[][3]){
  printf("%d\n",*(*(arr+2)+1));
  printf("%p\n",(*arr)+2);
  printf("%p\n",&arr[0][2]);
  printf("%d\n",*(((*arr)+1)+1));
}
int main(){
  int a[][3]={5,6,7,8,9,4,3,2,1};
  fun(a);
  return 0;
}
```

1. `printf("%d\n", *(*(arr + 2) + 1));`:
   - `arr + 2`: Moves the pointer to the third row: `[3, 2, 1]`.
   - `*(arr + 2)`: Dereferences to get the third row.
   - `*(arr + 2) + 1`: Moves to the second element in the third row.
   - `*(*(arr + 2) + 1)`: Dereferences to get the value `2`.
   - **Output**: `2`.
2. `printf("%p\n", (*arr) + 2);`:
   - `*arr`: Points to the first row: `[5, 6, 7]`.
   - `(*arr) + 2`: Moves to the third element in the first row.
   - **Output**: Prints the memory address of `&arr[0][2]`.
3. `printf("%p\n", &arr[0][2]);`:
   - Directly retrieves the memory address of the third element in the first row.
   - **Output**: Prints the memory address of `&arr[0][2]` (same as above).
4. `printf("%d\n", *(((*arr) + 1) + 1));`:
   - `*arr`: Points to the first row: `[5, 6, 7]`.
   - `(*arr) + 1`: Moves to the second element in the first row: `6`.
   - `((*arr) + 1) + 1`: Moves to the third element in the first row: `7`.
   - `*(((*arr) + 1) + 1)`: Dereferences to get the value `7`.
   - **Output**: `7`.

8. Explain the below declaration(s).

```c
(1) int process(int (*pf)(int a, int b)) ;
(2) int (*fun(int, void (*ptr)()))();
(3) int *(*p)(int (*a)[ ]);
(4) int (*p)[10];
(5) float *p[20];
(6) int p(char *a);
(7) int (*p(char * a))[10];
(8) int * (*p [10]) (char  *a);
```

7. `int (*p(char *a))[10];`

- **Explanation**:
  - `p` is a function that takes a single argument, a pointer to a character (`char *a`).
  - The function `p` returns a pointer to an array of 10 integers.

8. `int *(*p[10])(char *a);`

- **Explanation**:
  - `p` is an array of 10 function pointers.
  - Each function takes a single argument, a pointer to a character (`char *a`).
  - Each function returns a pointer to an integer (`int *`).

3. `int *(*p)(int (*a)[]);`
- **Explanation:**
  - `a` is a pointer to an array of integers.
  - `p` is a pointer to a function that takes `a` (a pointer to an array of integers) as an argument and returns a pointer to an `int`.

4. `int (*p)[10];`
- **Explanation:**
  - `p` is a pointer to an array of 10 integers.

5. `float *p[20];`
- **Explanation:**
  - `p` is an array of 20 pointers, where each pointer points to a `float`.

6. `int p(char *a);`
- **Explanation:**
  - `p` is a function that takes a single argument, a pointer to a character (`char *a`), and returns an `int`.

1. `int process(int (*pf)(int a, int b));`
- **Explanation:**
  - `pf` is a pointer to a function that takes two `int` arguments (`a` and `b`) and returns an `int`.
  - `process` is a function that takes this function pointer `pf` as a parameter and returns an `int`.

2. `int (*fun(int, void (*ptr)()))();`
- **Explanation:**
  - `ptr` is a pointer to a function that takes no arguments (`void`) and returns `void`.
  - `fun` is a function that takes two arguments:
    - An `int`.
    - A function pointer `ptr` as described above.
  - `fun` returns a pointer to a function that takes no arguments and returns an `int`.

9. What is printed by the following ANSI C program?

```
#include<stdio.h>
int main(void){
 int x = 1, z[2] = {10, 11};
 int *p = NULL;
 p = &x;
 *p = 10;
 p = &z[1];
 *(&z[0] + 1) += 3;
 printf("%d, %d, %d\n", x, z[0], z[1]);
 return 0;}
```

Output: `10, 10, 14`

Explanation:

1. `p = &x; *p = 10;` changes the value of `x` to `10`.
2. `p = &z[1];` points `p` to the second element of array `z`.
3. `*(&z[0] + 1) += 3;` increases `z[1]` (11) by `3`, making it `14`.
4. The final values: `x = 10`, `z[0] = 10`, `z[1] = 14`.

10. Find the output and different types of pointer involved

```
int main(){int *p=NULL;
  p=(int *)malloc(sizeof(int));
  *p=10;
  free(p);
  int *q;
  q=(int *)malloc(sizeof(int));
  *q=15;
  printf("%d  %d\n",*p,*q);
  return 0;}
```

Output: Undefined behavior. Possible: `0 15`.

Explanation:

1. `*p = 10; free(p);` frees the memory pointed to by `p`.
2. `q` is allocated new memory and assigned `15`.
3. Accessing `*p` after freeing results in undefined behavior.

```c
#include<stdio.h>
#include<stdlib.h>

void fun(int **q);
int main() {
    int *p = (int *)malloc(sizeof(int));
    *p = 55;
    fun(&p);
    printf("%d %p\n", *p, p);
    return 0;
}

void fun(int **q) {
    int r = 20;
    **q = r;
    printf("%p\n", *q);
}
```

Output:

```
20 <address>
20 <address>
```

Explanation:

1. `fun()` modifies the value of `*p` to `20` through the double pointer `q`.
2. `p` still points to the same address, but its value is updated.

12. Select the desire output of the following code snippet with reason;

```
int *fun();
int main(void){
  int *ptr;
  ptr=fun();
  printf("%d\n",*ptr);
  return 0;
}
```

```
int *fun(){
 int a=10,b=20;
 int sum=0;
 sum=sum+a+b;
 return &sum;
 }
```

Output with reason▼

(A) Unexpected behavoir
(C) 30
(B) Address of sum
(D) None of these

13. Select the desire output of the following code snippet with reason;

```
int *fun();
int main(void)
{
  int *ptr=fun();
  printf("%d\n",*ptr);
  return 0;
}
```

```
int *fun(){
 int a=10,b=20,*sum;
 sum=(int *)malloc(
     sizeof(int));
 *sum=a+b;
 return sum;
 }
```

Output with reason▼

(A) Unexpected behavoir
(C) 30
(B) Address of sum
(D) None of these

- Realloc allocates new location without loosing the data. Since the ptr is re-allocated , it did not lose the old value.

14. Find the output of the following program.

```
int main(){int *ptr;
 ptr=(int *)realloc(NULL,sizeof(int));
 *ptr=100;
 printf("%d\n",*ptr);
 return 0;}
```

Output: `100`

Explanation:

- `realloc(NULL, sizeof(int))` behaves like `malloc()` and allocates memory.
- The value `100` is stored and printed.

15. Write the output of the following program.

```
1  int main(){int *ptr;
2   ptr=(int *)calloc(1,sizeof(int));
3   *ptr=100;
4   printf("%d\n",*ptr);
5   ptr=(int *)realloc(ptr,0);
6   ptr=NULL;
7   printf("%p\n",ptr);
8   return 0;}
```

```
100
0x0
```

Explanation:

1. `calloc()` initializes memory to `0`, and the value is updated to `100`.
2. `realloc(ptr, 0)` deallocates the memory, similar to `free()`.
3. Setting `ptr = NULL` explicitly clears the pointer.

Q16)

```c
#include<stdio.h>
#include<stdlib.h>
int main() {
    int b = 65;
    void p = b; //O/t: Compile-Time Error
    printf("%d", p);
    return 0;
}
```

17. Select the output of the following program.

```
int main(){
 int b=65;
 void *p=&b;
 int *j=(int *)p;
 char *ch=(char *)p;
 printf("%d  %c\n",*j,*ch);
 return 0;
}
```

o/t: 65 A

18. Write the output of the code snippet. Also show the stac

```
int main(){int i;
 int *p=(int *)malloc(sizeof(int));
 *p=100;
 p=(int *)malloc(5*sizeof(int));
 for(i=0;i<5;i++){
    scanf("%d",p+i); /* 10,20,30,40,50 */
 }
 for(i=0;i<5;i++){
  printf("%d...%d\n",p[i],*(p+i));
 }
 return 0;}
```

Output: Depends on input. For input `10 20 30 40 50`:

```
10...10
20...20
30...30
40...40
50...50
```

Explanation:

- `p` is reallocated to hold 5 integers.
- Values are read into the array and printed using two equivalent expressions.

19. Write the output of the code snippet. Also show the st

```
int main(){
int i,*p,*rp;
p=(int *)malloc(5*sizeof(int));
for(i=0;i<5;i++)
   scanf("%d",p+i); /* 10,20,30,40,50 */
rp=(int *)realloc(p,10*sizeof(int));
for(i=5;i<10;i++)
   scanf("%d",rp+i);/* 9,8,6,5,4 */
for(i=0;i<10;i++){
 printf("%d...%d\n",rp[i],*(rp+i));
}
return 0;}
```

```
10...10
20...20
30...30
40...40
50...50
9...9
8...8
6...6
5...5
4...4
```

xplanation:

1. `malloc(5 * sizeof(int))` allocates memory for 5 integers.
2. The first 5 inputs are stored in `p`.
3. `realloc(p, 10 * sizeof(int))` resizes the memory block to hold 10 integers. The first 5 values remain, and the next 5 inputs are added.
4. Values are printed using both `rp[i]` and `*(rp + i)`.

20. Which of the following statements are true?.

```
(1) (void *)0 is a void pointer
(2) (void *)0 is a NULL pointer
(3) int *p=(int *)0; p is a NULL pointer
(4) a[i]==i[a]
(5) a[i][j]== *(*(a+i)+j)
```

Correct Statements:

1. **(1)**: True. `(void *)0` is a valid void pointer.
2. **(2)**: True. `(void *)0` is also considered a NULL pointer.
3. **(3)**: True. Casting `0` to a pointer type makes it a NULL pointer.
4. **(4)**: True. `a[i] == i[a]` because of the commutative property in array indexing (`*(a + i) == *(i + a)`).
5. **(5)**: True. `a[i][j] == *(*(a + i) + j)` due to pointer arithmetic.

21. State the output of the code.

```
#include<stdio.h>
int f(int n){
 while(--n>=0){
  printf("%d ",n-2);}
 return 1;}
```

```
int main(){
  int (*p)(int)=f;
  (*p)(8);
  return 0;}
```

[ma05-5]

```
5 4 3 2 1 0 -1 -2
```

Explanation:

1. `--n` decrements `n` from `8` to `0`.
2. In each iteration, `printf("%d ", n - 2);` prints `n - 2`.
3. Sequence:
   - When `n = 8`, `n - 2 = 5`.
   - When `n = 7`, `n - 2 = 4`.
   - ...
   - Ends when `n = 0` (printing `-2`).

```
void fun(){
   int *q=(int *)malloc(sizeof(int));
   *q=20;
}
int main(){
 int *p;
 int *r=NULL;
 fun();
 return 0;
}
```

Q22)

Statements:

1. **p is a wild pointer: True.** `p` is uninitialized and points to an undefined location.
2. **r is a NULL pointer: True.** `r` is explicitly set to `NULL`.
3. **q is a dangling pointer: True.** Memory allocated to `q` in `fun()` is not freed, and `q` goes out of scope after `fun()` ends.
4. **p is a dangling pointer: False.** `p` is not assigned any memory.
5. **fun() causes a memory leak: True.** The memory allocated by `malloc` is not freed, leading to a leak.

23. Check the error or output of the following program?

```
int main(){
void *p;
int *i=20;
p=&i;
void *q=p; //line-4
//line-5
printf("%d %d %d\n",i,*p,*q);
}
```

(i) 20 20 20
(ii) 20 30 20
(iii) compile error
at line-4
(iv) compile error
at line-5

Output:

Compile-time error

Explanation:

1. `int *i = 20;` is invalid because `20` is not a memory address.
2. Errors will occur at initialization and dereferencing.

24. Write the output of the given code snippet.

```
#include<stdio.h>
int main(){
   void demo();
   void (*fun)();
   fun=demo;
   (*)fun();
   fun();
   return 0;
}
```

```
#include<stdio.h>
void demo(){
 printf("SS");
}
```

SSSS

Explanation:

1. `fun = demo;` assigns the function pointer `fun` to the address of `demo`.
2. Both `(*fun)()` and `fun()` call the `demo()` function, printing `"SS"` twice.

25. Write the output of the given code snippet

```
int fun(int x,int y){
  int z=x+y+x*y;
  return z;
}

#include<stdio.h>
int main(){
  int (*fun_ptr)(int,int);
  fun_ptr=fun;
  int x=fun_ptr(34,56);
  printf("%d\n",x);
  return 0;
}
```

0/t: 1994 (By simple BODMAS Calculation)

26)

```
#include<stdio.h>
int main(){
int x,y;
int (*fun_ptr[2])(int,int);
fun_ptr[0]=fun1;
x=fun_ptr[0](4,5);;
fun_ptr[1]=fun2;
y=(*fun_ptr[1])(4,5);
printf("%d...%d\n",x,y);
return 0;
}
```

```
int fun1(int x,int y){
 return x+y;
}

int fun2(int x,int y){
 return x*y;
}
```

```
9...20
```

Explanation:

1. `fun_ptr[0] = fun1` assigns `fun1` to the first element of the array.
   - `fun1(4, 5)` returns `4 + 5 = 9`.
2. `fun_ptr[1] = fun2` assigns `fun2` to the second element of the array.
   - `fun2(4, 5)` returns `4 * 5 = 20`.
3. The results are printed as `9...20`.

27. Find out the correct syntal(s) for making a constant poin
and pointer cannot be modified).

```
(1) const <data_type> * ptr;
(2) <data_type> * const ptr;
(3) <dat_type> const *ptr;
(4) <data_type> const * const fun_ptr
(5) None of these
```

Correct Answer:

- (2): `<data_type> * const ptr;`
  - Declares a constant pointer, meaning the pointer's value (memory address) cannot be modified.

28. Find out the correct syntal(s) for a pointer to constant
of the variable/array that it points).

```
(1) const <data_type> * ptr;
(2) <data_type> * const ptr;
(3) <dat_type> const *ptr;
(4) <data_type> const * const fun_ptr
(5) None of these
```

Correct Answer:

- (1) and (3):
  - `const <data_type> * ptr;`
  - `<data_type> const *ptr;`
  - Both declare a pointer to a constant value, meaning the value at the address cannot be modified.

29. Select the correct way of declaring and initializing

```
(1) int (*ptr)(int,int,int)=funname;
(2) int *ptr(int,int,int)=funname;
(3) int (*ptr)(int,int,int)=&funname;
(4) (int *) ptr(int,int,int)=funname;
(5) None of these
```

Correct Answer:

- (1) and (3):
  - `int (*ptr)(int, int, int) = funname;`
    - Correctly declares a function pointer and assigns it the address of `funname`.
  - `int (*ptr)(int, int, int) = &funname;`
    - Explicitly uses the address of the function, which is equivalent.