Objective:

- The tasks given below will help you review the basic programming skills related to arrays, pointers, and dynamic memory allocation and logic obviously.

Task-1: Focus on Pointer and Alias:

A. Does the following function contain any logical or syntactical error? If yes, then describe briefly.

```
int * whatIsIt()
{
  int a=10;
  return &a;
}
```

B. Consider the declaration 'int *const * * const p;'. Circle the expression(s) below which can be treated as L-Value.

A. *p
B. **p
C. ***p
D. &p
E. p

C. Use 'memcpy' function to copy the contents of one integer array into another integer array.

For Example:

Assume we have two arrays 'a' and 'b', i.e.

```
int a[5] = {1, 2, 3, 4, 5};
int b[5]= {14, 12, 5, 14, 21};
```

We want to copy the contents of array 'b' into array 'a', so, after After Using 'memcpy', both array contents will become as follows

```
a    =    {14, 12, 5, 14, 21};
b    =    {14, 12, 5, 14, 21};
```

Note: Google 'memcpy' to know about how to use it.

D. Declare an alias named as 'ref' of the following identifier 'p'.

```
int a = 10;
int *const * p = & a;
```

E. Consider the code below:

```
void wow()
{
        int *p = new int;
        p = new int[50];
}
```

Which of the following type of error this code may produce?

- **A.** Null pointer
- **B.** Dangling pointer
- **C.** Illegal memory access
- **D.** Memory leakage
- **E.** None of the above

Task-2: Focus on Logic:
Write a function, which finds the last occurrence of the minimum value in an integer array.
For Example:
If the array contains 90, 12, 9, 78, 90, 9, 78, 90

Then function should return 5, as 9 is the minimum value with its last occurrence at index 5.

Task-3: Focus on Logic:
Write a function, which receives two sorted arrays and merge them into a third array in sorted form.
For Example:

Consider array a = { 10, 25, 37, 40, 50, 51, 55, 60 }
array b = { 2, 5, 26, 27, 29, 32, 40, 45, 70, 80, 85, 90, 95, 98 }
merging them in third array:
array c = { 2, 5, 10, 25, 26, 27, 29, 32, 37, 40, 40, 45, 50, 51, 55, 60, 70, 80, 85, 90, 95, 98 }

Function Prototype: int * mergeSortedArrays(int * array1, int size1, int * array2, int size2)

Task-4: Focus on Logic:

Write a function, which receives an array of integer and arrange the numbers in the received array in alternating order.

For example if the array is [a1, a2, a3, a4…], arrange the array such that b1 <= b2 >= b3 <= b4 and so on.

Sample Input Array:

3	5	7	8	4	9

Sample Output Array:

3	5	4	8	7	9

Task-5: Focus on Arrays:
Just read through the following article and if you feel any sort of difficulty in understanding the content then feel free to discuss with me.

Arrays Revealed

Introduction

Arrays are the built-in containers of C and C++. This article assumes the reader has some experience with arrays and array syntax but is not clear on

- Exactly how multi-dimensional arrays work,
- How to call a function with a multi-dimensional array,
- How to return a multi-dimensional array from a function,
- How to read and write arrays from a disc file.

How to Define Arrays

First, there are only one-dimensional arrays in C or C++. The number of elements is put between brackets:

```
int array[5];
```

That is an array of 5 elements each of which is an int.

```
int array[];
```

won't compile. You need to declare the number of elements.

Second, this array:

```
int array[5][10];
```

is still an array of 5 elements. Each element is an array of 10 int.

```
int array[5][10][15];
```

is still an array of 5 elements. Each element is an array of 10 elements where each element is an array of 15 int.

```
int array[][10];
```

won't compile. You need to declare the number of elements.

Third, the name of an array is the address of element 0

```
int array[5];
```

Here *array* is the address of array[0]. Since array[0] is an int, *array* is the address of an int. You can assign the name *array* to an int*.

```
int array[5][10];
```

Here *array* is the address of array[0]. Since array[0] is an array of 10 int, *array* is the address of an array of 10 int. You can assign the name *array* to a pointer to an array of 10 int:

```
int array[5][10];

int (*ptr)[10] = array;
```

Fourth, when the number of elements is not known at compile time, you create the array dynamically:

```
int* array = new int[value];
int (*ptr)[10] = new int[value][10];
int (*ptr)[10][15] = new int[value][10][15];
```

In each case *value* is the *number of elements*. Any other brackets only describe the elements.

Using an int** for an array of arrays is incorrect and produces wrong answers using pointer arithmetic. The compiler knows this so it won't compile this code:

```
int** ptr = new int[value][10];    //ERROR
```

new returns the address of an array of 10 int and that isn't the same as an int**.

Likewise:

```
int*** ptr = new int[value][10][15];    //ERROR
```

new returns the address of an array of 10 elements where each element is an array of 15 int and that isn't the same as an int***.

With the above in mind this array:

```
int array[10] = {0,1,2,3,4,5,6,7,8,9};
```

has a memory layout of

0 1 2 3 4 5 6 7 8 9

Whereas this array:

```
int array[5][2] = {0,1,2,3,4,5,6,7,8,9};
```

has a memory layout of

0 1 2 3 4 5 6 7 8 9

Kind of same, right?

So if your disc file contains

0 1 2 3 4 5 6 7 8 9

Does it make a difference whether you read into a one-dimensional array or a two-dimensional array? No.

Therefore, when you do your read use the address of array[0][0] and read as though you have a one-dimensional array and the values will be in the correct locations.

Passing Multi-dimensional Arrays to Functions

This array:

```
int arr[3][4][5];
```

can be passed to a function if the argument to func() is a pointer to a [4][5] array:

```
void func(int (* arg)[4][5], unsigned int x);

int main()
{
        int arr[3][4][5];
        func(arr, 3);
}
```

But if the func argument is a pointer to an [5] array of int, you need to pass the &arr[0][0]:

```
void func(int (* arg)[5], unsigned int x, int y);
int main()
{
   int arr[3][4][5];
   func(&arr[0][0], 3, 4);
}
```

But if the func argument is a pointer to an int, you need to pass the &arr[0][0][0]:

```
void func(int * arg, unsigned int x, int y, int z);
int main()
{
    int arr[3][4][5];
    func(&arr[0][0][0], 3, 4, 5);
}
```

As you omit dimensions, notice that you need to add arguments to func() since the "array-ness" is lost on the call. This is called *decay of array* and it occurs whenever an array is passed to a function. From inside the function all you see is an address and not an array. That forces you to pass the number of elements in the "dimensions".

Returning Multi-dimensional Arrays from Functions

Returning an array from a function only has meaning if the array was created by the function. Otherwise, no return is necessary since an existing array is passed by the address. However, if the function has created the array on the heap, you can return the address of element 0.

The problem here is that you can't use the function return type unless you

- return a type or
- return a pointer to a type.

That is, you cannot return a pointer to an array since an array is not a type. So, if you create an array of int you can return the array as an int*:

```
int* func(int arg)
{
   int* temp = new int[arg];
   return temp;
}
int main()
{
   int* arr = func(5);
}
```

This does not work when you create a multi-dimensional array:

```
int (*)[5] func(int arg)   // ERROR: Cannot return an array
{
   int (* temp)[5] = new int[arg][5];
   return temp;
}
int main()
{
   int (* arr)[5] = func(4);
}
```

In this case you would pass in the *address* of a pointer to an array of 5 int:

```
void func(int arg, int (**rval)[5])
{
   int (* temp)[5] = new int[arg][5];
   *rval = temp;
}
int main()
{
   int (* arr)[5] = 0;
   func(4, &arr);
   //arr is now a [4][5] array of int
}
```

Or, you can do the same using an alias

```
void func(int arg, int (* & rval)[5])
{
   int (* temp)[5] = new int[arg][5];
   rval = temp;
}
int main()
{
   int (* arr)[5] = 0;
   func(4, arr);
   //arr is now a [4][5] array of int
}
```

Finally, as was stated at the beginning of this article: There are no multi-dimensional arrays in C++.
Therefore, the function could just create a one-dimensional array of the correct number of elements and return the address of element 0. In this case, element 0 is a type and you can use the return type of a function to return a pointer to a type. Then the calling function could typecast the return so the array can be used with multiple dimensions:

```
int* func(int arg)
{
   int * temp = new int[arg];
   return temp;
}
int main()
{
   //This is arr[60]
   int* arr = func(60);

   //This is arr[12][5] --> 12 x 5 = 60
   int (*arr1)[5] = (int(*)[5])func(60);

   //This is arr[3][4][5] -> 3 * 4 * 5 = 60
   int (*arr2)[4][5] = (int(*)[4][5])func(60);

   //This is arr[1][3][4][5] -> 1*3*4*5 = 60;
   int (*arr3)[3][4][5] = (int(*)[3][4][5])func(60);
}
```