EEEE1042 - **Practical 7**
Arrays and dynamic memory allocation.

In this practical, we will practice creating arrays and passing them to functions and then repeating this using dynamic memory management. You can perform the entire practical in a single `.cpp` file, but keep each portion separated and delimited from the others. There are three parts to Section 1 and three parts to Section 2. It is suggested that you use a single `main()` function and declare all the necessary subfunctions in a **well-delimited** part of your `main()`. Delimit the subsections from each other with comments `/******/` and with `printf()`'s to enable you to easily distinguish the output of one part from another.

1 Passing Arrays to functions

In this first part of the practical, we will create functions for printing 1D, 2D and 1D-as-2D arrays. 1D-as-2D means it is declared as 1D, but interpreted as 2D using the formula described in the lectures and also described below.

1.1 Create and print 1D array.

1. **Create a main function and declare an array** `x` of `10 int`s. Populate it with any values of your choice. Recall your array variable is a pointer to the beginning of the array.

2. **Create a function** `printArray1()` that takes 2 inputs:

   (a) A pointer to an `int` (which is your array)

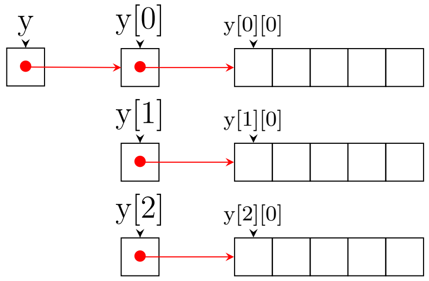
   (b) An int (which is the length of your array)

   Have your `printArray1()` function print out the array together with the index number in 2 columns.

3. Inside the `main()`, **pass your pointer** `x` **together with its length to the function** `printArray1()` and make sure your function is able to print it out.

1.2 Create and print 2D array (pointer-to-pointer).

1. **Create a 2D array** inside your main as a pointer-to-pointer. This can be done with: `int y[3][5]={{...},{...},{...}};` where you fill in your own 3 arrays of `int`s. Recall from the lecture notes that `y` is a pointer to an array of `int`s.



2. Now we are going to pass your pointer (to pointers) `y` to a function called `printArray2()` that will print the 2D array for you. **Declare the function `printArray2()` as**:
`void printArray2(int y[3][5], int M ,int N);`
This declares the first input to `printArray2()` to be a pointer to an array of 3 pointers (to arrays of size 5), which is equivalent to a `3x5` 2D array. `M` and `N` are the dimensions of the 2D array (3 and 5 respectively) that are to be used in the double-for loop during printing. `M` is the number of rows while `N` is the number of columns in the 2D array. The function could also be declared:
`void printArray2(int y[][5], int M ,int N);`
The first index in the declaration is not necessary as the compiler is able to count the number of elements in that array, and allocate the memory accordingly.

3. **Write the calling line for `printArray2()`** in the main function and make sure you are able to call it to print the 2D-array. Since `y` in the calling line is declared to be a pointer to an array of 3 arrays (of 5 `int`s), the input matches what is expected in the declaration and the compiler should allow the function call.

1.3 Create 1D array, interpret as 2D array.

As mentioned during the lecture, it is also possible to pass in a 1D array and calculate the offsets into that array yourself such that it behaves like a 2D array.

1. **Write another function** that takes in a 1D array (representing a 2D array) of `int`s and the dimensions of the 2D array `M` and `N`:
`void printArray3(int *x, int M, int N);`
Use the 1D array and the mapping `x[i][j]`$\Leftrightarrow$ `x[i*N+j]` to print out the 2D-array.

2. **Call `printArray3()` from inside the main function** after defining a new 1D array `z` holding 15 elements that are to be printed out as a `3x5` array. Make sure your function is able to process (in this case print) the 1D array as a 2D array.

2 Dynamic memory management.

In this part of the practical, we are going to dynamically allocate memory, fill it with values and pass it to the functions you created in part 1 above.

2.1 Allocating and printing 1D array of `ints`.

First we will dynamically allocate and print a 1D array of `ints`.

1. **Declare a pointer to an `int`**
   `int *a;`
   At this point, it is just a pointer that doesn't point to anything useful. You can't print it, you can't add or subtract to it.

2. Allocate memory to an arrary of 15 elements using `malloc()`:
   `a=(int*)malloc(15*sizeof(int));`
   At this point `*a` now points to something useful: allocated memory. But recall that memory allocated using `malloc()` is uninitialized.

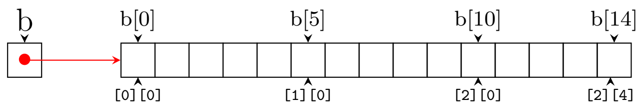
3. It is a good habit to immediately include the `free()` command after `malloc()` or `calloc()` if it is your intention to free this memory with:
   `free(a);`
   This way you won't unintentionally create a program that has memory leaks. **All the ensuing code should go between the `malloc()` and `free()` commands.** If you put it after the `free()`, you will be using memory that is no longer allocated, leading to segmentation faults.

4. Fill your array of `ints` with random numbers

5. Send your array of random numbers to `printArray1()` above. You should see it print out your dynamically allocated 1D array of `ints`.

2.2 Allocating 1D and printing as 2D array of `ints`.

The situation we will be setting up in this part of the practical is depicted by this figure:



1. **Create a new pointer to an array** of `ints`:
   `int *b;`
   At this point, it is just a pointer that doesn't point to anything useful. You can't print it, you can't add or subtract to it.

2. **Dynamically allocate some new memory using** `malloc()` for 15 `ints` and store the pointer to that memory in `b`:

`b=(int*) malloc(15*sizeof(int));`
Now the pointer is pointing to allocated, but uninitialized memory.

3. As above include the `free()` command after `malloc()` to avoid potential memory leaks.
`free(b);`
Remember to put all subsequent code **before** the `free()`

4. **Fill your 15 bytes of allocated memory with random integers**. You may randomize your seed with `srand()` if you so choose.

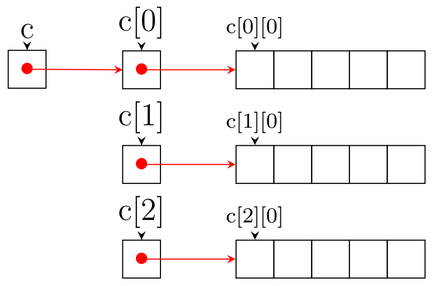
5. **Pass your pointer to the function** `printArray3()` defined above to print out your 1D array in 2D, make sure you can print out your `3x5` array.

6. **Reallocate the memory to a block of size** `4x5`:
`b=(int*)realloc(b,20*sizeof(int));`
This command asks the operating system to dynamically reallocate your previously allocated memory to be of size 20 bytes. You can now print out that block using `printArray3()` again.

7. **Print out your new** `4x5` **array** by passing it to `printArray3()`:
`printArray3(b,4,5);`
Notice that the first 15 bytes should be the same as before. `realloc()` will retain any values in the array that existed there before the reallocation. Any new values allocated by `realloc()` will be uninitialized.

2.3 Allocating and printing 2D array of `ints`.

1. Now we will dynamically allocate a 2D array as an array of pointers to pointers. The situation we will be setting up using dynamic memory management is:



Declare a pointer to pointer to an integer:
`int **c;`
At this time, the pointer is uninitialized and cannot be used. We will initialize it using calloc.

2. **Allocate your array of pointers**:
`c = (int**) calloc(3,sizeof(int*));`
This command allocates the array of three "pointers-to-integers" `c[0]`,`c[1]` and `c[2]`. Recall that `calloc()` returns pointer to `void` which is then type-cast as `(int**)`

(pointer-to-pointer-to integers) before assigning it to `c`. At this point `c` is allocated, but `c[0]`,`c[1]` and `c[2]` are still unallocated.

3. **Allocate the memory for each pointer `c[i]`**. This needs to be done in a for loop:
   `for (i=0;i<3;i++) c[i] = (int*)calloc(5,sizeof(int));`
   Each pointer-to-integers is now being allocated an array of integers of size 5. It too is typecast as `(int*)` (pointer-to-integers) before assigning it to the respective pointer `c[i]`. At this point the array of array of integers is set-up, but it doesn't have any values in it. You now need to populate your 2D array.

4. **Use a double-for loop, to put random values into your `3x5` array of integers**. You can index the values as `c[i][j]`. At this point you will have a completely allocated, and populated 2D array of integers ready for printing.

5. Unfortunately, the previous function `printArray2()` can't be used to print out an `int**` type. It was defined to print out an array of pointers to arrays of size 5, which is actually a pretty limiting category of inputs that it can take. Indeed, passing statically defined arrays to functions is not a common occurence in C programming. Here we will define a function `printArray4()` that can handle the more common situation of taking in a pointer-to-pointer. **Declare and define `printArray4()` as:**
   `void printArray4(int **x, int M, int N)`
   `x` is the pointer to pointer to an `int` array and `M` and `N` are the number of rows and columns in `x` respectively. The function definition itself is virtually the same as in `printArray2()`, it's just that the input type is defined differently. Now call `printArray4()` passing in your pointer-to-pointer and the array dimensions and make sure that it is able to print out your 2D-array (array of arrays).

6. Free'ing your 2D array is also more involved with double arrays. The free'ing must occur in reverse order to which it was allocated. You cannot `free(c)` before having individually free'd each `c[i]`. First you must free each `c[i]`:
   `for (i=0;i<3;i++) free(c[i]);`

7. At this point `c` is now not pointing to anything important, you can now free it too:
   `free(c);`
   `c` is now no longer pointing to anything of significance. It is a good idea to let future possible calls to `realloc()` know this by pointing `c` to NULL
   `c=NULL;`
   Although this is not needed here, as the pointer is no longer being used before the program ends, if at any point in the future, `c` is passed to a `realloc()` function, its behaviour could be unpredictable (and undesirable) if `c` is pointing to something that is not allocated. By setting it to NULL, it is now not pointing to anything. A call to `realloc()` when the pointer is NULL, behaves just like `malloc()`, so any future call to `realloc()` will now behave properly.

A sample output of your program could look as follows:

```
**************
Part 1.1
printArray1:
0 5
1 32
2 7
3 4
4 2
5 5
6 0
7 9
8 1
9 2
```

```
**************
Part 1.2
printArray2:
5 4 3 2 1
5 6 7 8 9
0 -1 -2 -3 -4
```

```
**************
Part 1.3
printArray3:
5 4 3 2 1
5 6 7 8 9
0 -1 -2 -3 -4
```

```
**************
Part 2.1
printArray1:
0 42
1 19
2 39
3 39
4 45
5 9
6 16
7 38
8 13
9 27
10 23
11 31
12 18
13 25
14 47
```

```
**************
Part 2.2
printArray3:
18 12 14 2 12
0 4 2 16 3
8 2 2 19 4
Part2.2: realloc()
printArray3:
18 12 14 2 12
0 4 2 16 3
8 2 2 19 4
0 0 0133393 0
```

```
**************
Part 2.3
printArray4
15 25 18 8 19
15 14 29 8 23
15 23 12 26 8
```

Save your program to the file: `yourName_practical7_EEEE1042.cpp` and submit it to Moodle. Be sure you have properly commented your code. When you are done with the practical, you may use the remainder of the session to work on your coursework.