Exercise dynamic memory allocation

1.) Consider you want to dynamically allocate memory for an array of arrays, where each array length is variable. The type of the arrays would be float. Write a function that takes two items: i) an integer P that represents number of arrays, and ii) An array called Lengths of size P that contains Lengths for P number of arrays. After allocating memory for the arrays, the function should fill-up the arrays with random numbers from 1 to 100 (you can put an int to a float variable. The integer will get promoted to float automatically). At the end the function returns appropriate pointer. Next write a set of statements to show how you would free-up the memory.

In the blank bellow, write the appropriate return type.

a) float ** AllocateArrayOfArrays(int P, int *Lengths) {

```
float **arrayOfArrays = (float**) malloc (P * sizeof(float*));
int i;
int j;

for(i=0; i<P; i++)
{
  arrayOfArrays[i] = (float*) malloc(Lengths[i] * sizeof(float));
}
for(i=0; i<P; i++)
{
  for(j=0; j< Lengths[i]; j++)
  {
    arrayOfArrays[i][j] = (rand() % 100) + 1;
  }
}
return arrayOfArrays;
```

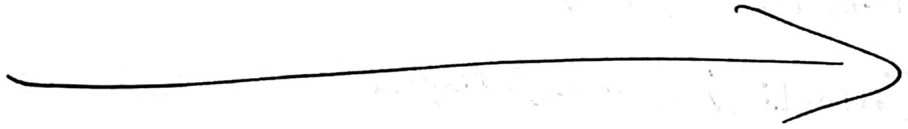
} // NOTE: srand can be used if time.h is included...

b) Write few C statements to declare an appropriate variable and call the AllocateArrayOfArrays() with example parameters. Later on write necessary C statement(s) to free the allocated memory.

↙ main function

← free function

Continued on back...



```
int main()
{
    int i;
    int j;
    float** floatArray;
    int rowLength = 5; // adjustable row size...
    int ColumnLength Array = 5; // adjustable column size...

    for (i=0; i<5; i++) // for loop < column size...
    {
      columnLength Array[i] = i+1; // assigns array elements to i+1, so
                                   // [0]=1, [1]=2, etc. ...
    }

    floatArray = Allocate Array Of Arrays (rowLength, ColumnLength Array);

    for(i=0; i<rowLength; i++)
    {
      for(j=0; j<ColumnLength Array[i]; j++)
      {
        // print statements can go here...
        printf("%f \n", floatArray[i][j]);
      }
    }
    freeArray (floatArray, rowLength);

    return 0;
}
Void freeArray (float **array, int rows)
{
    for (int i=0; i< rows; i++)
    {
      free(array[i]); // free(*array)...
    }
    free(array);
}
```

2. This problem relies on the following struct definition:

```
typedef struct Employee
{
        char *first; // Employee's first name.
        char *last; // Employee's last name.
        int ID; // Employee ID.
} Employee;
```

Consider the following function, which takes three arrays – each of length n – containing the first names, last names, and ID numbers of n employees for some company. The function dynamically allocates an array of n Employee structs, copies the information from the array arguments into the corresponding array of structs, and returns the dynamically allocated array.

```
Employee *makeArray(char **firstNames, char **lastNames, int *IDs, int n)
{
        int i;
        Employee *array = (Employee*) malloc( n * sizeof(Employee*) );
        for (i = 0; i < n; i++)
        {
                array[i].first = malloc( sizeof(char*) * n );
                array[i].last = malloc( sizeof(char* * n );
                strcpy(array[i].first, firstNames[i]);
                strcpy(array[i].last, lastNames[i]);
                array[i].ID = IDs[i];
        }
        return array;
}
```

a) Fill in the blanks above with the appropriate arguments for each malloc() statement.

b) Next, write a function that takes a pointer to the array created by the makeArray() function, along with the number of employee records in that array (n) and frees all the dynamically allocated memory associated with that array. The function signature is as follows:

```
void freeEmployeeArray(Employee *array, int n)
{
  for (int i=0; i<n; i++)
  {
    free(array[i]);
  }
  free(array);
}
```

3. Write two differences between malloc and calloc.

Both can allocate memory, but only calloc can initialize the memory on the same line of code. This also means that calloc takes 2 argument parameters instead of just 1.

4. What is the purpose of realloc?

realloc() is used to resize memory that has been previously allocated.

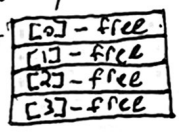
5. Write few lines of code that creates two arrays with malloc. Then write a statement that can create a memory leak. Discuss why you think your code has a memory leak by drawing the status of the memory after you use malloc and the line of the code you claim that creates a memory leak.

```
float **array2D = (float**) malloc (2 * sizeof (float *));
for (int i=0; i<2; i++)
{
    array2D[i] = (float *) malloc (3 * sizeof (float)); // [2][3] array...
}
float *arrayBasic = (float*) malloc (4 * sizeof (float));

free (arrayBasic);
free (array2D);
// the float* values within the
// float **array2D remain unfreed...
```

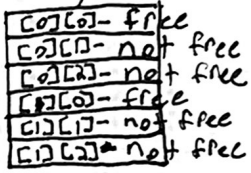
Memory after freeing...

arrayBasic:



X

array2D:



6. This question relies of the following structure definition.

```
struct Student {
        int student_id;
        float *quizzes;
}
```

Complete the following function that takes 2 int indicating number of students N and number of quizzes Q. The function allocate appropriate memory for N Students and for each student it should allocate memory for Q quizzes. Then take input for all the data and return the pointer.

```
struct Student* AllocateStudents( int N, int Q) {
    struct Student* students = (struct Student*) malloc (N * sizeof (struct Student));

    for(int i=0;  i<N;  i++;)
    {
       Students[i].quizzes = (float*) malloc (Q * sizeof (float));
    }
    for(int i=0; i<N; i++) //begin user input data retrieval...
    {
      printf("Enter id: ");
      scanf("%d", &students[i].student_id);
      for(int j=0; j<Q; j++)
      {  printf("Enter quiz data: ");
         scanf("%f", &students[i].quizzes[j]);
      }
    }
    return students;
}
```

Consider you are calling the function by the following statement:

```
struct Student *students = AllocateStudents(5, 2);
free_up_memory (students, 2);
//write this function on the right side to free up the memory
```

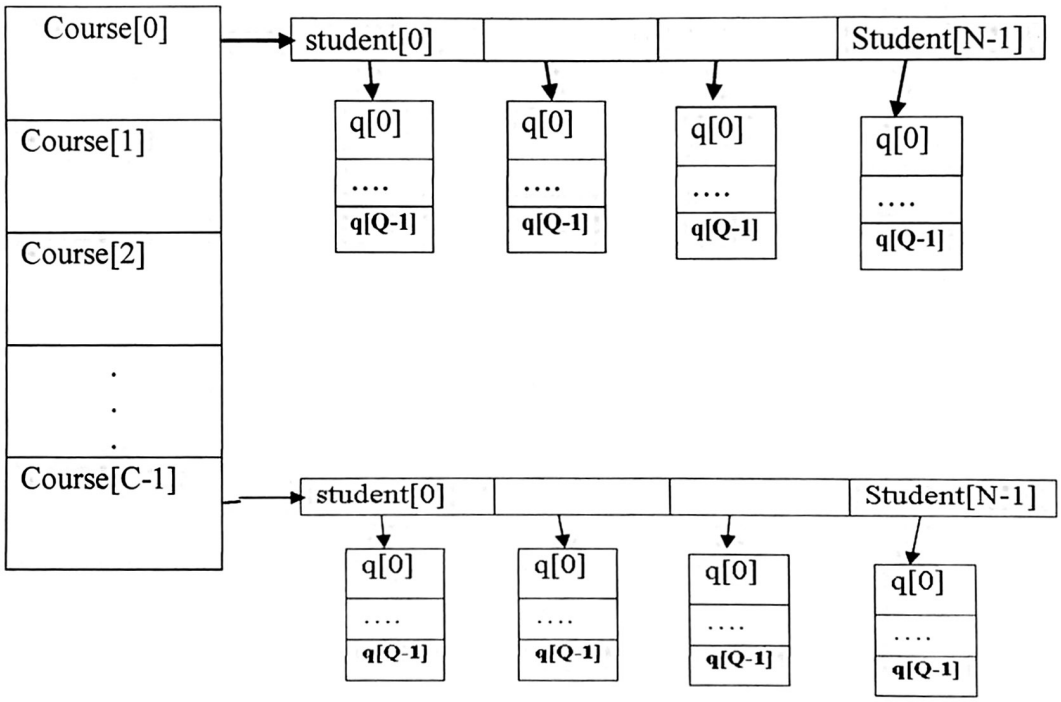
```
Void free_up_memory(struct
    Student* students, int N)
{
  for(int i=0; i<N; i++)
  {
    free(students[i].quizzes);
  }
  free(students);
}
```

7.
This question relies of the following structure definition.

```
typedef struct Student {
        int student_id;
        float *quizzes;
} Student;
```

There are C number of courses and each course has N number of students and each student has to take Q number of quizzes.

You can visualize it by the following picture.



a) Write a function that takes C, N, and Q and returns appropriate memory to store the data. //You don't have to fill-up the memory with any data. Just allocate the memory.

```
Student **  AllocateCourse_Students(int C, int N, int Q){
    Student ** courses = (Student **) malloc (C * sizeof (Student *));
    for (int i=0; i<C; i++)
    {
      courses[i] = (Student *) malloc ( N * sizeof (Student));
      for(int j=0; j< N; j++)
      {
        courses[i][j].quizzes = (float *) malloc (Q * sizeof (float));
      }
    }
  return courses;
}
```

b) Write a function to free-up the memory. Provide appropriate parameter for the function.

On back

```
void free_up_memory(Student** courses, int C, int N) // int Q not needed..
{
  for(int i=0; i<C; i++)
  {
    for(int j=0; j<N; j++)
    {
      free(courses[i][j].quizzes); // free quizzes...
    }
    free(courses[i]); // free students...
  }
  free(courses); // free courses...
}
```