**LANGARA COLLEGE**

*DEPARTMENT OF COMPUTING SCIENCE AND INFORMATION SYSTEMS*

CPSC 1160 - ALGORITHMS AND DATA STRUCTURES I

Assignment 03 – Lab 03

September 28, 2017

**Instructions**

* This assignment is worth 10 points, and is due on October 5 at 04:00 PM.
* Download this file, edit it to include your answers, name it **Lab03**, and upload it on D2L.
* Clarify your answers by providing code examples if you find it more helpful.

**Multidimensional Dynamic Arrays (Mapping Method)**

During this week, you learned how to use a dynamically-allocated block of data to store and manipulate the elements of a multi-dimensional array. Let’s call this method “*Mapping*”.

**Question 1 [2 points]**

What are the advantages and disadvantages of mapping a multidimensional array into a dynamically-allocated (single-dimensional) array? Explain your answer.

In general, in computer science when it comes for storing multidimensional arrays in linear storage, row-major order and column-major order are possible methods to do so. The difference between these orders lies beneath the contiguous order of stored elements of an array in memory. Row-major order is used to C based languages like C++. While these terms imply a matrix, these orders can be generalized to any array, by noting that the terms row-major and column-major are equivalent to lexicographic (lexical, alphabetical, mathematical, amongst others) orders. Data layout and design is crucial for accurately passing arrays between software written in different computing languages. It is also important for performance sequential reading for CPUs process is more efficiently than no sequential data. To represent this within a single dimensional array, you multiply the row index by the width and then add the column index:

    int arrayList[width \* height];

    int setElementsOfTheArray(int row, int col, int value) {

        value = array[width \* row + col];

    }

Well these concepts are quite similar for the dynamic-allocated memory but with important distinctions. With the keyword “new” you will request an address of row times columns amount of memory so there isn’t any waste of memory (like previous concepts used before in which we demanded a MAX\_SIZE constant value to store our array wasting a certain amount or having to create a new one in case of overflow) on unused elements in the Array. Those variables will be running in the program until the programmer manually flush them through the keyword “delete” or the program reach an end. Nevertheless, the non-linear search will create a speed efficiency issue and it will be an initial extra cost of memory for pointer allocation.

//  We do Dynamic memory allocation for efficiency reasons

#include <iostream>

using namespace std;

int \*matrix; // matrix here will be only a pointer, an address

int main() {

    int rows, cols;

    cin >> rows >> cols;

    matrix = new int[rows \* cols]; // this request an address of rows\*columns amount of memory for allocation

    // It may be more convenient row by row

    for (int i = 0; i < rows; i++) {

        for (int j = 0; j < cols; j++) {

            cin >> matrix[(i - 1) \* cols + j]; // needed mapping for using single index

        }

    }

    int rowIndex, colIndex;

    cin >> rowIndex >> colIndex;

    cout << matrix[rowIndex][colIndex];

    system("PAUSE");

    delete []matrix; // flush the allocated memory

    return 0;

}

// ((i - 1) \* cols) + j - 1 if the count started from 1

**Multidimensional Dynamic Arrays (Array-of-Arrays Method)**

Multidimensional arrays are also defined as an array of arrays. This fact can be utilized to use an array of pointers or the pointer-to-pointer concept to dynamically allocate memory to a multidimensional array. Watch a very helpful video on this topic from [here](http://www.youtube.com/watch?v=i5gUlnrUqqQ) and answer the following question.

**Question 2 [2 points]**

What are the advantages and disadvantages of creating a dynamic array of (dynamic) arrays to implement a multidimensional array? Explain your answer.

Knowing the context within the previous question this creates flexibility because we no longer need to know the initial value of the array size at compile time. Nonetheless, we will be forced to use pointer to a pointer in some cases (2D) or more depending on the number of dimensions using more initial memory for pointers memory allocation. This layer process allows to represent any number of dimensions we might need

**Static Arrays vs. Mapped Dynamic Multidimensional Arrays vs. Dynamic Array of Arrays**

Now, you have three options to manipulate a multidimensional array: static arrays, mapped dynamic arrays and dynamic array of arrays. Answer the following questions accordingly.

**Question 3 [2 points]**

Compare the above-mentioned methods from the viewpoint of efficiency.

**Static array** is a special case of Fixed-length array. In this case, the length not only can be changed but also can be determined at runtime. Local arrays are allocated on the stack, and have automatic storage duration, they got destroyed when the function in which they’re created ends. They necessarily have a fixed size:

    int foo[10];

They will be read in a linear search so it won’t waste time jumping from pointer to pointer (more activation records) but they will waste a lot of memory for indexes never used and they will need contiguous memory in stack for allocation. If you are developing desktop / mobile architectures (Intel / ARM 32 / 64 bit processors) locality also matters (cache memory). Your variables will be accessed faster if they are already stored in the cache. When it comes to the *Stack* is always the winner since is so frequently used that it is very likely to always sit in the cache memory. So, small well known arrays are always best allocated there.

Using true multi-dimensional arrays instead of composing one from pointers, may also help on this matter, since true arrays are always a linear chunk of data stored in memory, so it usually might need smaller portions of blocks of cache memory to load in.

**Dynamic arrays** are arrays types created with the operator new[], they have dynamic storage duration and are stored on the heap. They can have any size (so they are way more flexible), but you need to allocate and flush them manually since they are not a part of the stack frame itself:

    int\* foo = new int[10];

    delete[] foo;

Both mapped dynamic arrays and dynamic array of arrays will use pointers with the distinction of the last will use several pointers equivalent to the number of dimensions desired using extra cost of memory allocation in the Heap. A scattered pointer composition like this, might need more cache blocks, and may rise cache line conflicts depending on how these chunks of memory are physically constructed and finished on the heap.

**Question 4 [4 points]**

Which of these methods are more suitable to implement a ragged (jagged) array? How would you use the other two methods if you have to do so? Explain your answers

We may have 2 possible approaches:

One should be creating a 2D matrix (involving vectors) that is equal in dimension to the largest Array dimension. So let’s say if the Array is an array of pointers of length N and the maximum dimension of any individual vector is M, then create a model C(N,M) and fill each row of C with a vector of the Array. This method, while requiring extra storage, will likely yield the fastest access.

Another method would be creating the following format: (assuming an array of integer elements):

int\*\* arr = new int\*[5];

for(size\_t i = 0; i < 5; ++i) {

    arr[i] = new int[4];

}

and this generates a two-dimensional dynamically allocated array of 5 by 4. It can be used like this:

arr[i][j] = 15;

Then de-allocate the memory after you are done:

for(size\_t i = 0; i < 5; ++i) {

    delete[] arr[i];

}

delete[] arr;

Tomas Gonzalez Ortega – 100266942 – Langara College