Exam 2 Review - Fall 2023

Exam Details

- The exam is made up of:
    - Writing code to solve problems.
    - Tracing execution given specified input values.
- There are multiple versions of the exam.
- **Partial credit is available** for every question.
    - Your goal should be to not need it.
    - Focus your solution on correctness of the problem solving method. Clear evidence of your thought process and plan must be included in the code and comments in order to qualify for partial credit.
    - Put down anything you do know even if you can't solve the entire problem. Some partial credit is better than getting no credit.
    - Make your thinking visible so applying partial credit is easier.
- **Not allowed**
    - Any electronic devices (including calculators, phones, smart watches, and computers)
- **Allowed**
    - A writing utensil (e.g. pen/pencil)
    - Scratch Paper (8.5X11 inches)
    - Up to 5 pages of exam aids that you create.
        - Exam aids can be pages up to 8.5X11 inches and can be handwritten or printed on both sides.
        - You can't use exam aids for scratch work during the exam.
            - If you do, then you must submit the exam aid with your exam.
- We will be manually grading the exam, so don't get stressed if it is not perfect since you will get partial credit. We will also ignore minor errors that do not significantly impact the results.

Exam Strategies

- Understanding at a high level will help you code better during an exam. So, as you study, ask yourself...
    - *What is happening?*
        - *High level: What is the goal?*
        - *Code level: What are the effects and consequences of statements, etc.?*
    - *Why is it that way?*

        - *High level: Why are certain design decisions made? Why is it designed that way?*
        - *Code level: What is the code that accomplishes a higher-level goal? Why is it done that way?*
    - *How is it done?*
        - *High level: How do you accomplish a high-level goal?*
        - *Code level: How do you write the code?*
- Study Strategy: Prioritize how you study to maximize your ability to demonstrate concept mastery while not overwhelming yourself.
    - For a solid passing grade focus on class topics, slides, examples, in-class activities, homework, and labwork.
        - Make sure you understand anything coded in homework, labwork, and in-class examples.
            - What general (higher level) principles were illustrated in a labwork/homework?
                - Could you explain it in general? (i.e. what/why rather than how)
                - Be able to solve and write code for similar problems.
    - After you've done all you can do in the "for a solid passing grade" above, then focus on zyBook Challenge Activities and Participation Activities.
        - If a challenge or activity seems fuzzy, review the text for clarification.
        - Key terms are bolded.
    - After you've done all you can from the previous approaches and if you are still fuzzy about a topic, read about it in a different source or textbook. You can also talk to someone who can explain it in a different way.

Exam Topics

Problem solving requires accumulative knowledge of all topics covered in the course. *This exam will focus on:*

Parameter passing

- Pass by value
- Pass by reference (a variable or a pointer variable)
- const modifier

Dynamic Memory

- Operations
    - Allocation
    - Deallocation
    - Access
- Where allocated
- Memory Management
    - Garbage Collection
    - Memory Leaks: definition, prevention strategies

Dynamic 1D and 2D Arrays

- create (declare, define/allocate, initialize)
- read (access, search)
- update (insert, remove, resize)
- destroy (deallocate)
- traversal
- resize
- Tracking size of array
    - Size variable
    - Sentinel value

Practice Problems

These practice problems are meant to help you review concepts related to the exam. You may find more practice problems by doing optional parts of the labworks that you did not work on, reviewing examples from your lecture material, and studying other examples that may be shared on Canvas from other instructors.

PathLength

Write a function with declaration

`double pathLength(double** distance, int n, int* path, int m)`
where

- `distance` is a `n` by `n` 2d-array such that position `distance[i][j]` stores the road distance in miles from city `i` to city `j`;
- `path` is an integer array with `m` elements that stores a sequence of cities visited in a trip, i.e., `0 <= path[i] < n` for all `i` such that `0 <= i < m`
- `n` and `m` are greater than zero.

The function should return the length in miles of the path.

For example, for n = 5, distance matrix below

0.0	30.0	10.0	70.0	10.0
30.0	0.0	45.0	100.0	50.0
10.0	45.0	0.0	85.0	20.0
70.0	100.0	85.0	0.0	100.0
10.0	50.0	20.0	100.0	0.0

and m = 3, path

0	1	2

pathLength(distance, n, path, m) returns 30 + 45 = 75.

For same n and distance, but with m = 6 and path

0	1	0	3	2	0

pathLength(distance, n, path, m) returns 30 + 30 + 70 + 85 + 10 = 225

AvgMatrix

Write a function with declaration

```
void avgMatrix (double** inArray, int rows, int columns, double** outArray)
```

that gets a 2d-array `inArray` with `rows` >= 2 rows and `columns` >= 2 columns with double elements and calculates `outArray` as the average matrix, i.e., `outArray` is an `rows` by `columns` matrix where each element is obtained by the average of the (at most 8) neighbor elements in `inArray`.

Example:

For the 3 by 4 matrix:

0.5	2.0	1.2	3.0
-1.0	1.5	3.0	2.4
0.0	1.0	1.5	2.0

the `outArray` would be:

0.833	1.04	2.38	2.2
1.0	1.025	1.825	2.14
0.5	1.0	1.98	2.3

Minus Odd Column

Write a function with declaration

```
void minusOddColumn(int** mat, int n)
```

where n >= 1 and `mat` is an `n` by `n` 2d-array of non-negative integers.

The function should find the column in `mat` with the most odd numbers and replace all of its elements with -1. If multiple columns have the highest presence of odd numbers, the first of them (i.e., the one with the lowest index) should be chosen to have its elements replaced by -1.

For example, for `n = 5` and mat:

0	30	10	70	10
30	0	45	100	50
10	45	0	85	20
70	100	85	0	100
10	50	20	100	0

the function will result in mat

0	30	-1	70	10
30	0	-1	100	50
10	45	-1	85	20
70	100	-1	0	100
10	50	-1	100	0

For n = 2 and `mat`

0	30
30	0

the function will result in `mat`

-1	30
-1	0

get_even_numbers

Write a function that receives an array **A** of **n** integers (all elements >= 0) and returns a dynamically allocated array containing all the even elements in A. It also returns in the parameter **m** the number of elements in the created array.
The signature of the function is:

```
int* get_even_numbers(int* A, unsigned int n,
                      unsigned int& m);
```

Separate even from odds

Write a function that receives an array **A** of **n** integers (all elements >= 0) and returns a dynamically allocated array containing all even elements of A first, then all odd elements of A. The signature of the function is:

```
int* separate_even_odd(int* A, unsigned int n);
```

Array ordered by frequency

Write the function and the main program as specified below:

a. write a function with the following signature:

```
int* get_array_ordered_by_frequency (int *a,
                    unsigned int size_a
                    unsigned int& new_array_size);
```

that, given an array **a** of **size_a** elements, it returns a dynamically allocated array such that the new array contains all elements of a (without repetition), ordered by how frequently they appear in a.
Function parameters:

i. **a** is an array of integers

ii. **size_a** is the number of elements in array a

iii. **new_array_size** is the number of elements in the array being created by the function

The function returns the address of a dynamically created array such as the new array contains the numbers that appear in **a,** ordered by frequency

Example:
For array a = {-1, 3, -1, 1, 1, 4, 3, 3}, the function returns array {3, -1, 1, 4} or {3, 1, -1, 4} (since -1 and 1 appear the same number of times) and new_array_size will be 4.

b. write the main program that reads **n**, an array of **n** integers, and prints the list of numbers that appear in the given array, ordered by how frequently these numbers appear in the input array. Use arrays that are dynamically allocated and make sure your program is not leaking memory.

Remove all-zeros rows

Write a function that, given a 2D array, returns a new 2D array containing only the rows that have at least one non-zero element in the row.

```
int** remove_allzeros_rows(int** matrix,
                           int nrows,
```

```
                                int ncolumns,
                                int& new_nrows);
```

If `nrow` or `ncolumns` is zero, throw an exception.

Increase Array

Write a function that, given a 2D array and the number of rows and columns to add, resize the array to a size that is the number of previous rows and cols plus the indicated number of rows and columns. First initialize new cells on a row to be the sum of the cell to the left, to the upper left, and the lower left. Next initialize new cells in a column to the sum of the cell above, above and left, and above and right.

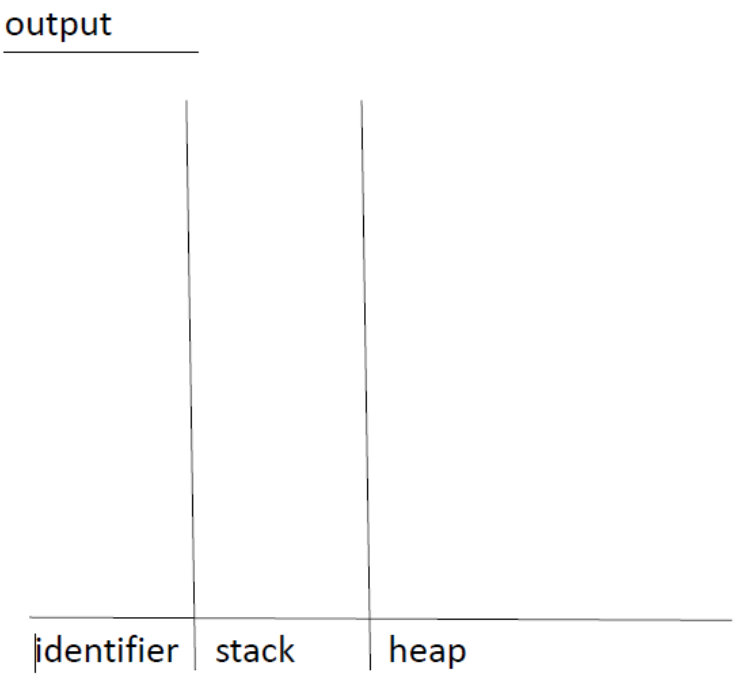
```
void increaseArray(int**& ary, int& numRows, int& numColumns,
                   int addRows, int addColumns);
```

Trace Code

Trace the execution of the following program and show how the values evolve in memory.

- Indicate where in memory each variable exists. I.e. stack or heap
- It should be clear which function has each variable.
- Show the values of variables over time.
    - So don't erase prior values, mark through prior values and write new values next to it.
    - You should be able to see every value any variable had throughout the program.
- Show all of your steps that lead to the final output. Minor errors can lead to an incorrect result.
- Ways to show your thinking:

- The memory diagram format we have used in class is excellent for showing the information.



- Any work that shows how you got to your answer.

Program to trace

```
#include <iostream>
using std::cout;
using std::endl;

int sideEffectSum(int c, int& d) {
    c *= 3;
    d /= 2;
    return  c + d;
}

int swapIfEvenSum(int& a, int& b) {
    if ((a+b)%2 == 0) {
        int temp = a;
        a = b;
        b = temp;
    }
    return sideEffectSum(a, b);
}

int heapSum(int* e, int* f) {
    return *e + *f;
```

```
}


int main()
{
    int z = 12;
    int y = 7;

    int *w = new int(swapIfEvenSum(z, y));
    z++;
    int *v = new int(swapIfEvenSum(z, y));

    int u = heapSum(w,v);

    cout << u << " " << *v << " " << *w << " ";
    cout << y << " " << z << endl;

    delete w;
    delete v;

    return 0;
}
```