CS103: Introduction to Programming
Spring 2023 – Midterm 2 Exam
03/30/23, 7 PM – 8:40 PM

[Complete all the information in the box below.]

Name:__Solutions_____

Student ID: _____ **Email:** _____**@usc.edu**

Lecture section (Circle One):

Redekopp	Redekopp	
MW 12:30 p.m.	MW 2:00 p.m.	

Ques.	Max score	Time
1	2	3 min.
2	7	8 min.
3	11	14 min.
4	8	15 min.
5	10	25 min.
6	12	35 min.
Total	50	100 min

1. Dynamic Memory True/False (2 pts)

For each of 1.1 – 1.4, indicate **TRUE** for each statement below that uses proper syntax (i.e. will compile) and **FALSE** otherwise. Assume **n** and **m** are properly initialized integers.

1.1. **True/False**: `string* x = new string[m];`

1.2. **True/False**: `char* x = new string;`

1.3. **True/False:** `double** mat = new double*[m];`

1.4. **True/False:** `int** mat = new int[n][m];`

2. Multiple Choice (7 pts)

Choose and circle the correct option.

2.1. **True/False**: In C++, **struct**s cannot have member functions.

2.2. **True/False**: If `x` is a **pointer** to a `vector<int>` object, then `x.push_back(3)` will correctly add 3 to the back of the vector. Would need to be `x->push_back(3)`

2.3. **True/False**: This code would have memory errors and/or segfault.

```
vector<int> x;
for(int i=0; i < 10; i++)
   { x[i] = 0; }
```

2.4. **True/False**: A vector can add a value to the front in O(1).

2.5. **True/False**: A doubly-linked list with a **head** and **tail** pointer can serve as a double-ended queue (deque) data structure and meets its runtime requirements.

2.6. `ifstream` objects generally support the same operations as what other object(s) [choose one]:

a) cout
b) string
c) cin
d) vector

2.7. **True/False**: Given your knowledge of the **string** type and appropriate memory management, each time a string variable is assigned a new value, a **`new`** and **`delete`** operation may take place (e.g. `string x = "hi"; x = "bye bye";`)

3. **Classes (11 pts):** Study the following code (split over 2 pages) which attempts to model a CS Lab, the TAs assigned to lead that lab, and determine who will lead the lab each week.

```
06  // Intentionally starts at line 6 - assume appropriate #includes/using statement above
07  class Leader {
08  public:
09      Leader(string n) { name = n; }
10      int lead() { return ++timesLed;  }
11      string name;
12  private:
13      int timesLed;
14  };
15
16  class CSLab {
17  private:
18      deque<Leader> leaders;
19      Leader* currLdr;
20      void addLeader(string leaderName) {
21          Leader theLeader(leaderName);
22          leaders.push_back(theLeader);
23      }
24  public:
25      CSLab() { currLdr = /*__ Blank 1 __*/; }        // Ctor 1
26      CSLab(string leaderName) {                      // Ctor 2
27          addLeader(leaderName);
28          currLdr = &leaders[0];
29      }
30      CSLab(deque<Leader> theLeaders) {               // Ctor 3
31          leaders = theLeaders;
32          currLdr = &leaders[0];
33      }
34      ~CSLab()  { /* Empty for now */ }               // Dtor
35      void runLab();                                  // Defined on the next page
36  };
```

3.1. What is the single keyword/expression that should be in **blank 1** in **Ctor1**? **_NULL____**

3.2. **True / False:** Calling `addLeader()` on line **27** will compile (is visible and accessible).

3.3. If the line `CSLab lab1;` was written in `main()`, which `CSLab` constructor would be called?

a. **Ctor 1** b. Ctor 2 c. Ctor 3 d. None of the above

3.4. If the line `CSLab lab2("TATina", "TATommy");` was written in `main()`, which `CSLab` constructor would be called?

a. Ctor 1 b. Ctor 2 c. Ctor 3 d. None of the above

3.5. **True / False:** Given a `CSLab` object declared as `CSLab lab1;` in `main()`, the expression: `lab1.addLeader("TATina");` is valid and will compile.

```
37  void runLab() {
38      if( !currLdr ) {
39          cout << "No lab today – no leader" << endl;
40          return;
41      }
42      cout << "Ldr: " <<  currLdr->name << endl;
43      cout << "Times led: " << currLdr.lead() << endl;
44      Leader t = leaders.front();
45      leaders.pop_front();
46      leaders.push_back(t);
47      currLdr = &leaders[0];
48  }
```

3.6. When compiling the code, the compiler indicated that in `runLab()`, `leaders` and `currLdr` were "`undefined in this scope`". The root cause of this error is on what line of code (**6-48**)? *[Note: for the remaining problem assume this error is fixed].*

Line Number: ___**37**__ (Just put a single integer line number in the blank)**.**

3.7. **True / False:** In **runLab()** on line **42**, the use of `currLdr->name` will compile (i.e. uses the right syntax and **name** is visible and accessible).

3.8. **True / False:** In **runLab()** on line **43**, the use of `currLdr.lead()` will compile (i.e. uses the right syntax and **`lead()`** is visible and accessible).

3.9. **True / False:** `runLab()` could correctly be declared as a **`const`** member function (e.g. `void runLab() const`).

3.10. If there are **n** leaders in the `leaders_` deque, the runtime of `runLab()` is:

a. **O(1)** b. **O(n)** c. **O(n^2)**

3.11. **True / False:** If the size of the `leaders_` deque is **1**, the operations on lines **44-46** will cause a segmentation fault.

3.12. **True / False:** Given the implementation of the **`CSLab`** class shown above, it requires additional code in the destructor to avoid memory errors (as found by valgrind).

For 3.13-3.15, review the code of the **`Leader`** class on the first code listing. When running a program that used it, valgrind will report which kind of error(s)? *[mark all that are correct as True]*.

3.13. **True / False:** Out-of-bounds (invalid) read error will be reported.

3.14. **True / False:** Access to uninitialized memory will be reported.

3.15. **True / False:** Memory leak will be reported.

4. **Coding 1 (8 pts)** – Write a function that takes an array of integers of length **n**, and should return a pointer to a valid integer array that holds ONLY the integers from the original array that are **even** (in the same relative order as they appeared in the original array). It should also output the size of this new array by setting the value pointed to by **psize** with the size.
Example: Given an input array of: `0,1,2,3,4` return an array with `0,2,4` and set the integer pointed to by **psize** to **3** (since that is the size of the returned array)**.**
Note 1: The output array cannot be larger than needed but must fit the exact amount of even integers
Note 2: Return NULL and set the integer pointed to by `psize` to **0** if no even integers exist or the input array is of size 0 and ensure your code works for any input array size.

```
//  orig     input array (containing event and/or odd integers)
//  n        size of the input array
//  psize    pointer to an integer to be set with the output array length
//  return:  Pointer to output array with only even values, or NULL if no
//                even numbers were present.
int* evennums(int orig[], int n, int* psize)
{   // Your code here
    // Note: can't assume any specific starting value for *psize

    // Need to first count how many even numbers exist
    int numeven = 0;
    for(int i=0; i < n; i++){
        if(orig[i] % 2 == 0){
            numeven++;
        }
    }
    // Need to check for 0 even numbers because ...
    if(numeven == 0){
        *psize = 0;  // Need to set *psize (not psize)
        return NULL;
    }
    // ... it would be wrong to allocate a 0-size array
    int* retval = new int[numeven]; // requires dyn. alloc.

    int k = 0; // index for output array
    // Now copy only the even integers into the newly created array
    for(int i=0; i < n; i++){
        if(orig[i] % 2 == 0){
            retval[k] = orig[i];
            k++;
        }
    }
    *psize = numeven;  // Need to set *psize (not psize)
    return retval;
}
```

5. **Coding 2 (10 pts)** – You are given a **singly-linked list class** that contains data members for both the **head** and **tail** pointers. Write a member function: **push_back_if_not_present** which adds a new item/value at the **end of the list** ONLY if it is **NOT already present** in the singly-linked list. Return **false** if the item was already present or **true** if it was not and you added it to the back.

Example 1: So if the list had: **1 2 3**, calling `list.push_back_if_not_present(-2)` would add -2 : **1 2 3 -2** and return **true.**

Example 2: So if the list had: **1 2 3**, calling `list.push_back_if_not_present(2)` would not change the list and return **false.**

```
struct Item {
    int val;
    Item* next;
};
class ListInt {
public:
    ListInt();
    ~ListInt();
    bool empty() const;
    void push_front(int new_val);
    void pop_front();
    bool push_back_if_not_present(int x);
    void print() const;
private:
    Item* head;  Item* tail;
};
```

You may NOT call other member for your implementation of `push_back_if_not_present()`, and **may NOT add data members**. Your solution should **visit Items once (i.e. run in O(n))**, **leave the list in a correct state for future operations, and not produce memory errors**.

Write your code on the next page.

```
bool ListInt::push_back_if_not_present(int x) {
    // Case 1: empty
    if(head == nullptr){
        Item* newptr = new Item;
        // Will be the "last" (and "first") element so use
        // next pointer equal to null
        newptr->val = x; newptr->next = nullptr;
        head = newptr; // set head to newptr
        tail = head; // or newptr
    }
    // Case 2: 1 or more items
    else {
        // APPROACH 1 (EASIER) - Walk full list, then use tail to add item
        //=====================================
        Item* temp = head; // don't lose your head
        // should check if value exists by walking the list
        while(temp != NULL){
            // Case 2a: x IS ALREADY present
            if( temp->val == x ){
                return false;
            }
            temp = temp->next;
        }
        // Case 2b: x is not present...add it
        //  We can use tail to add it.
        Item* newptr = new Item;
        newptr->val = x; newptr->next = nullptr;
        tail->next = newptr;
        tail = newptr;

        // // APPROACH 2 - Not using tail; walk TO last item checking as you go.
        // //=====================================
        // Item* temp = head; // don't lose your head
        // // walk until last/tail item, checking as we go
        // while(temp->next != NULL && temp->val != x){
        //     temp = temp->next;
        // }
        // // Case 2a: x is ALREADY present
        // // Note: this will also check the last element's value if we
        // //  stopped because we reached the last element.  Could
        // if(temp->val == x){
        //     return false;
        // }
        // // Case 2b: x is not present...add it
        // else {
```

```
        //      Item* newptr = new Item;
        //      newptr->val = x; newptr->next = nullptr;
        //      temp->next = newptr; // or tail->next = newptr
        //      tail = newptr;
        // }
    }
    return true;
}
```

6. **Coding 3 (12 pts)** – An instructor of a class that has only **homeworks** and **exams** has all of the students' assignment scores mixed together in a single text file using the format shown below (usernames, category title, and score) in the sample input file (**grades1.in**)

```
ttrojan  hw  100
yyortsos  hw  90
ttrojan  exam  87
cfolt  exam  83
cfolt  hw  91
ttrojan  hw  85
yyortsos  hw  82
last  exam  -1
```

The instructor would like a program that takes filename from the command line as well as a student name (e.g. `./grades grades1.in ttrojan`) and then reads the data in the file to populate a Student object with that students username, HW, and exam scores. The Student object is shown below and should also provide member functions to compute the average HW grade and average exam grade (assuming all HWs are equally weighted for the HW grade and all exams are equally weighted for the Exams grade).

```
struct GradeItem {
    int cat; // 0 = HW, 1 = Exam
    int score;
    // Convenience constructor
    GradeItem(int c, int s) { cat = c; score = s; }
};
class Student {
public:
    Student(string uname);
    bool initGrades(char* filename);
    double getAvg(int which) const;
private:
    string uname_;
    vector<GradeItem> items_; // HW and Exam scores for this student
};
```

Let us refer to each score (and it corresponding category) for a student as a grade item. The file may contain ANY number of grade items...there is no upper limit. But processing of grade items should end when a grade item that contains **last** as the username and **-1** as the score is encountered (and that record should then be discarded). Note: We recommend against use of getline() as reading the input can be done more simply.

You may use C++ `string` and `vector<T>` (where T is any type) in this program as you like. But **no other C/C++ libraries may be added beyond those already included below**. We have also defined a `GradeItem` struct that you must use to store each score for the given student.

We have provided a code skeleton with constructor and other functions, including the majority of `main()`. Complete the **`initGrades()`** and **`getAvg()`** member functions of the Student class and then **fill in the missing lines in `main()`** where indicated.

- Read the comments in the code which act as requirements you must meet.
- You may not alter other aspects of the skeleton that is given.
- Your code should not be blatantly inefficient (e.g. do in $O(n^3)$ what could be done in $O(n^2)$).

```
#include <iostream>
#include <fstream> // for ifstream
#include <string>
#include <vector>
using namespace std;  // no other libraries may be added

struct GradeItem {
    int cat;   // 0 = HW, 1 = Exam
    int score;
    GradeItem(int c, int s) { cat = c; score = s; }  // Convenience constructor
};
class Student {
public:
    Student(string uname) { uname_ = uname };
    bool initGrades(char* filename);
    double getAvg(int cat) const;
private:
    string uname_;
    vector<GradeItem> items_; // HW and Exam scores for this student
};

int main(int argc, char* argv[]) {
    if(argc < 2) {
        cout << "Please provide a grades file" << endl;
        return 1;
    }
    string username(__argv[2]_____); // may only fill in one expression
    Student s1(username);
    // Call `initGrades(...)` on the s1 object passing the appropriate argument

    __s1.initGrades(argv[1])_____________________________________;

    cout << "HW Avg: " << s1.getAvg(0) << endl;
    cout << "Exam Avg: " << s1.getAvg(1) << endl;
    // Any cleanup code if necessary – leave blank if none is necessary
    ____<should be left blank - none necessary>_____________________________
    return 0;
}  // write initGrades() and getAvg() on the next page
```

```
// Complete your code - Return false if any errors occurred opening or
//                      reading from the file, true otherwise
bool Student::initGrades(char* filename)
{   // Reminder: Only add the grades of the corresponding student
    ifstream ifile(filename);
    if(ifile.fail()){
        cout << "Couldn't open file" << endl;
        return false;
    }
    string name, cat;
    int score;

    ifile >> name >> cat >> score;
    // Use DeMorgan's correctly. Should be what is shown below or
    // (!(name == "last" && score == -1)).  Or they can check in the loop
    // and break, if done correctly.
    while(name != "last" || score != -1){
        if(name == uname_){  // Is this record/item for THIS student?
            if(cat == "hw") {
                GradeItem item(0, score); // create HW item
                items_.push_back(item);
            }
            else {
                GradeItem item(1, score); // create Exam item
                items_.push_back(item);
            }
        }
        ifile >> name >> cat >> score; // get next input record/item
    }
    if(ifile.fail()){
        return false;
    }
    ifile.close();  // not strictly necessary but good practice
    return true;
}
// Complete your code - Should return 0 if no scores for the given category exist
double Student::getAvg(int cat) const
{
    double tot = 0;
    int count = 0;
    for(unsigned i=0; i < items_.size(); i++){
        if(cat == items_[i].cat ){
            tot += items_[i].score;
            count++;
        }
    }
    if(count == 0) return 0.0; // handle case of 0 HWs or Exams
    else return tot/count;
}
```