Computer Science Fundamentals II

Practice MIDTERM #1

Spring 2020

You will have fifty minutes to complete an exam like this. There are five problems on six pages. You'll be asked to *write your answers* on blank paper that I will provide. Examining the entire practice exam questions now, I'd say that the exam is collectively harder than an exam I would normally give, but hopefully they will serve as good practice as each of the problems *is* like something I'd expect you to be able to answer on an exam. I'd expect anyone to complete at least four of these problems in 50 minutes.

You are not to use any resources while completing Friday's exam. Complete each problem to the best of your ability. Each problem requests that you devise a program or a fragment of a program's code in C++. You should do your best to produce working code—syntactically correct and runnable—that meets the specification of the problem.

If for some reason you forget some aspect of C++ syntax, your best option is to alert us to that fact (e.g. "I can't remember the operator for integer division in C++; I am using '//' to mean that.") letting us know what you had intended. You may find it useful to include explanation of your code, or comments, but it's not necessary that you do so.

You are welcome to write additional functions or other supporting pieces of code that get used by the code we ask you to write, should you feel the need to do so.

1. **Write a C++ function** `primes` that takes a positive integer `n` and gives back the pointer of an array of `n` integers allocated on the heap. That array should hold the first n prime numbers. The algorithm you use should check primeness by relying on the list of primes it has generated so far. For example, when fed 6 and if `primes` has filled the array with 2, 3, 5, 7 so far, then it should determine that 8, 9, and 10 are not prime by only checking amongst those numbers as possible divisors. When that call completes it should return the vector with the sequence 2, 3, 5, 7, 11, 13. The function should begin its work by allocating the array and putting the value 2 in the 0th item of the array. It should then loop to fill out the remaining `n-1` items in the array.

2. The admissions, registrar, and alumni offices are working on new database code, written in C++, and so they've enlisted CS2 students. **Give the definition** of a struct `ReedStudent` that has four pieces of information

   - `nameTag`: a string representing the way that student should be addressed. This string is normally their common name, such as `"Jim Fix"`.
   - `status`: a string representing their status, one of `"first year"`, `"sophomore"`, `"junior"`, `"senior"`, or `"graduate"`.
   - `year`: an integer year of school (like, say, 2019) for tracking their progress in Reed's courses. This will be the year they arrived if they are a first year, and the year they last completed a spring semester if they have higher status.
   - `isTransfer`: a boolean of whether or not they were a transfer student.

   Using this definition, **define a function** with the following C++ declaration

   ```
   ReedStudent attendOneYear(ReedStudent s) { ... }
   ```

   It should return a `ReedStudent` struct with some changes to `s` as follows:

   - The returned struct should have an increased `status` by one year (e.g. a junior becomes a senior), unless they are already a graduate. They graduate if they are a senior.
   - It should increase their year by 1, regardless of their status. (Alumni can still attend classes after they've graduated, and so the registrar would like to keep tracking this.)
   - Their `nameTag` should stay the same, unless they are graduating. If they are graduating then a string should be appended to `nameTag`, and this is described below.

   Changing their name tag: If they graduate with `attendOneYear`, then their graduating class year should be appended to their `nameTag` string. For example, if they graduate this coming year—that is, they switch from `"senior"` to `"graduate"` when their `year` changes from 2019 to 2020—then the string appended should be `", B.A. '20"`. As another example, had I been a senior at Reed last year, I would graduate with a call to `attendOneYear`, and then my `nameTag` would read `"Jim Fix, B.A. '19"`.

   To do this appending of the graduation date, you'll need to call `std::to_string` to convert an integer to a string. In using this, it's okay to assume that their graduation year is between 2010 and 2099. That is to say: you can assume that the tens digit of their graduation year is not 0.

3. Draw the stack and heap for the following C program just before it returns 0:

```
int main() {
  int x = 6;
  int* y = &x;
  int* z = new int[1];
  y[0] = 10;
  z[0] = x;
  x++;
  return 0;
}
```

4. Below are definitions of two C++ structs to define a linked list data structure, one that stores a collection of integers.

```
struct node {
  int data;
  struct node* next;
};

struct {
  node* first;
}
```

Here is a picture of a `list` of type (`llist*`) storing the sequence $\langle 10, 37, 5 \rangle$.

Recall the design of linked lists, specifically how to navigate their structure. For example, the value of `list->first->data` is 10, the value of `list->first->next->data` is 37, and the value of `list->first->next->next->data` is 5. Furthermore, the pointer value of `list->first->next->next->next` is equal to `nullptr`. If instead the list was empty, then `list->first` would be `nullptr`.

Write the C++ code for a function `void reverse(LList * list)`. It should restructure the nodes of the linked list it is given so that the nodes are linked in an order that is the reverse of their order before `reverse` was called. **The code should not modify the** `data` **field of the nodes, nor should it allocate any new nodes, nor an array, nor any other data structure.** *It can just traverse the list and change each of the* `next` *fields of the nodes.*

If `reverse` was called with the linked list depicted above, it would have this structure upon the function's return:

5. A matrix is a two-dimensional array of floating point values. It has a number of rows and a number of columns. Below left is a picture of a 3×4 matrix. A *transpose* is a flip of a matrix along its top-left to bottom-right diagonal. The right picture gives the transpose of the matrix to the left. The rows become the columns, and vice versa.

0.1	0.1	0.4	0.3
0.1	0.5	0.6	0.5
0.4	0.6	0.8	0.7

Fig. A matrix with 3 rows and 4 columns.

0.1	0.1	0.4
0.1	0.5	0.6
0.4	0.6	0.8
0.3	0.5	0.7

Fig. The transpose of that matrix.

Below is the definition of two C++ structs to define a matrix:

```
struct matrixRow {
    double *col;
};

struct matrix {
    matrixRow *row;
    int rows;
    int cols;
};
```

A `matrix` struct knows its number of `rows` and its number of `cols`. It also has a pointer `row` to an array of `matrixRow` structs. Each `matrixRow` struct has a pointer `col` to an array of `double` values. The picture at the bottom left of this page shows a matrix struct `*m` laid out in heap memory. Note that, for that pictured example, `m->row[0].col[3]` holds the value 0.3. The picture at the bottom right of this page shows the transpose of the matrix. For it, `mt->row[3].col[0]` holds the value 0.3 instead.

Write a C++ function `matrix *transpose(matrix *m)` that creates a new matrix object with a new array of rows, and where each row is a new array of doubles. It should return a pointer to this new matrix, and that matrix should be the transpose of the matrix `m`.

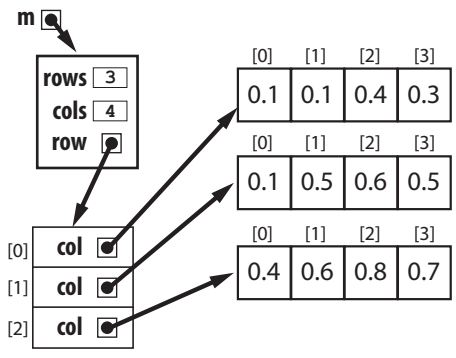

Fig. A `matrix *m`.

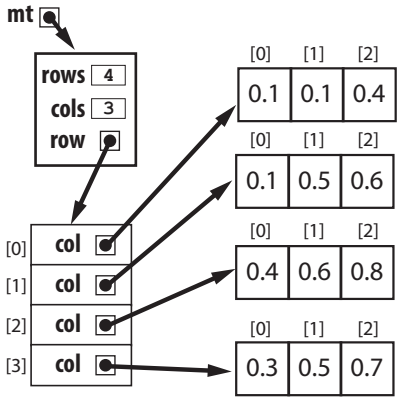

Fig. A `matrix *mt = transpose(m)`.