

CS106X Practice Midterm

Exam Facts:

When: Wednesday, November 4th at 7:00 – 9:00 p.m.

Where: Gates B03

Coverage

The midterm covers everything up through today's lecture and the material addressed by Assignments 1 through 5. We will not be especially picky about syntax or other conceptually shallow ideas. We are looking for a clear understanding of the core programming concepts in C++.

The exam is open-book and open-notes, but no computers allowed. Don't let the open-book nature mislead you. There isn't enough time to learn or even **re**-learn everything during a two-hour window. You should be experienced enough with the material to readily answer the questions, relying on your notes only for the occasional detail.

Writing code on paper in a relatively short time period is not quite the same as working with the compiler and a keyboard. I recommend that you practice writing out solutions to these practice problems—starting with a blank sheet of paper—until you're certain you can write code without a computer to guide you.

SCPD students can either come to campus, or they can take the exam remotely anytime after Wednesday night at 7:00 p.m. The exam will be posted as a handout once it's been administered, at which point you can pull it, print it, give yourself an uninterrupted block of two hours to take it, and then fax it in to me when you're done. **My cell and fax numbers will be printed on the front page of the exam.** You can call me during if you have questions, and you should use the fax number to get us a copy of your exam. Hold on to your original until you get the faxed copy back. You don't need to take the exam at the office, and you don't need a proctor. Just self-administer so you have maximum flexibility on when you take it. SCPD students can take the exam on Thursday if need be, provided I have your faxed copy by Thursday at 5:00 p.m. We'll be grading on Thursday night, so we need your exams by then.

Problem 1: Acronyms

An acronym is an abbreviation formed from the first letter of each of a series of words, such as NATO (North American Treaty Organization) or HTML (Hyper Text Markup Language). In this problem, you are to write two functions that operate on a table of acronyms. An acronym data file contains a list of expanded acronyms, one per line, such as shown here:

```
American Automobile Association
Internal Revenue Service
Standard Query Language
Parent Teacher Association
Animal Acupuncture Academy
Inertial Reference System
Abdominal Aortic Aneurysm
National Direct Student Loan
```

The terms within each expansion are separated by at least one and possibly more spaces. All terms within an expansion are capitalized and thus the acronym formed is uppercase.

- a) The **ReadIntoMap** function reads an acronym data file and builds a map from acronyms to expansions. The two parameters to the function are a correctly opened **ifstream** and an empty map. The function should fill the map with entries where acronym (e.g. "**IRS**") is the key, and the associated value is its expansion(s) (e.g., "**Internal Revenue Service**" and "Inertial Reference System"). An acronym may have more than one expansion (e.g., there are three options for "AAA" in the above file) and thus the value for each key is a vector, where each entry is a possible expansion. A Scanner will be useful here.

```
void ReadIntoMap ifstream& in, Map<Vector<string> >& map)
{
```

- b) Although using acronyms can be convenient, it can be confusing when one acronym has many different expansions. Given a map such as the one created by the function from part (a), the **PercentConfusing** function returns the percentage of acronyms in the map that have more than one expansion. For a map constructed from the data file shown on the previous page, there are five unique acronyms, of which two have multiple expansions, so the returned percentage would be 40% (expressed as a double that's **0.40**).

```
double PercentConfusing(Map<Vector<string> >& map)
{
```

Problem 2: Pascal's Travels

You're given a grid of positive integers to represent a game board, where the $[0, 0]$ entry is the upper left corner. The number in each location is the number of squares you can advance in any of the four primary compass directions, provided that move doesn't take you off the board. You're interested in the total number of distinct ways one could travel from the upper left corner to the lower right corner, given the constraint that no single path should ever visit the same location twice.

2	5	3	1	5	4	3	2
3	1	2	3	1	4	5	3
2	3	3	4	1	3	2	2
4	5	1	1	2	5	2	1
2	4	4	4	3	3	2	1
2	5	5	1	2	2	1	4
4	3	2	1	4	1	3	3
4	1	4	4	2	4	2	5

Consider the initial game board to the left, and notice that the upper left corner is occupied by a 2. That means you can take either two steps to the right, or two steps down (but not two steps to the left or above, because that would carry you off the board). Suppose you opt to go right so that you find yourself in the configuration to the right.

2	5	3	1	5	4	3	2
3	1	2	3	1	4	5	3
2	3	3	4	1	3	2	2
4	5	1	1	2	5	2	1
2	4	4	4	3	3	2	1
2	5	5	1	2	2	1	4
4	3	2	1	4	1	3	3
4	1	4	4	2	4	2	5

From there, you could continue along as follows:

1

2	5	3	1	5	4	3	2
3	1	2	3	1	4	5	3
2	3	3	4	1	3	2	2
4	5	1	1	2	5	2	1
2	4	4	4	3	3	2	1
2	5	5	1	2	2	1	4
4	3	2	1	4	1	3	3
4	1	4	4	2	4	2	5

2

2	5	3	1	5	4	3	2
3	1	2	3	1	4	5	3
2	3	3	4	1	3	2	2
4	5	1	1	2	5	2	1
2	4	4	4	3	3	2	1
2	5	5	1	2	2	1	4
4	3	2	1	4	1	3	3
4	1	4	4	2	4	2	5

3

2	5	3	1	5	4	3	2
3	1	2	3	1	4	5	3
2	3	3	4	1	3	2	2
4	5	1	1	2	5	2	1
2	4	4	4	3	3	2	1
2	5	5	1	2	2	1	4
4	3	2	1	4	1	3	3
4	1	4	4	2	4	2	5

4

2	5	3	1	5	4	3	2
3	1	2	3	1	4	5	3
2	3	3	4	1	3	2	2
4	5	1	1	2	5	2	1
2	4	4	4	3	3	2	1
2	5	5	1	2	2	1	4
4	3	2	1	4	1	3	3
4	1	4	4	2	4	2	5

5

2	5	3	1	5	4	3	2
3	1	2	3	1	4	5	3
2	3	3	4	1	3	2	2
4	5	1	1	2	5	2	1
2	4	4	4	3	3	2	1
2	5	5	1	2	2	1	4
4	3	2	1	4	1	3	3
4	1	4	4	2	4	2	5

6

2	5	3	1	5	4	3	2
3	1	2	3	1	4	5	3
2	3	3	4	1	3	2	2
4	5	1	1	2	5	2	1
2	4	4	4	3	3	2	1
2	5	5	1	2	2	1	4
4	3	2	1	4	1	3	3
4	1	4	4	2	4	2	5

So, this series of moves illustrates just one of potentially several paths you could take from upper left to lower right. Your job here is to write a function called **NumPaths**, which takes a **Grid** of integers and computes the total number of ways to travel to the lower right corner of the board. Note that you never want to count the same path twice, but two paths are considered to be distinct even if they share a common sub-path. And because you want to prevent cycles, you should change the value at any given location to a zero as a way of marking that you've been there. Just be sure to restore the original value as you exit the recursive call. You'll want to write a wrapper function and some utility functions to decide if you're on the board or not.

```
int NumPaths(Grid<int>& board)
{
```

Problem 3: Sanitizing Strings

Write a recursive procedure called **Sanitize**, which takes a string of text and a **Vector<string>** of banned substrings and returns the shortest string that can be generated by eliminating all illegal substrings. For instance, if the only banned substring is **abc**, then the string **babaabc**bc**** can be sanitized down to **b**, via:

```

babaabcbc
bababc
babc
b

```

Note that each string in the stack is the same as the one above it, except that the underlined substrings have been spliced out.

A more elaborate example illustrates that **baacabacaaabca**abb**aa** can be edited down to **baa** if the set of banned substrings includes **ac**, **ab**, and **caa**, as illustrated by:

```

baacabacaaabcaabbaa
baacabacaaabcaabaa
baacacaaabcaabaa
baacaabcaabaa
baabcaabaa
baabcaaa
bacaaa
babaaa
baa

```

Write a recursive procedure called **Sanitize** that computes and returns the **smallest** string that can be generated by an optimal series of substring eliminations.

Recall that the **string**'s **find** method searches the receiving string from **start** forward and returns the smallest index where **substring** can be found (or **string::npos** if **substring** isn't present.)

```

int string::find(string substring, int start = 0);
string Sanitize(string str, Vector<string>& substrings)
{

```

Problem 4: Stretching Lists

Write a function called **Stretch** that takes a linked list of positive integers and stretches the list. Each node **n** gives birth to a sequence of **n.value - 1** identical nodes and splices them in right after the original. Assume that all integers are positive.

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

Here are some examples:

list	list after call Stretch(list)
1 → 4 → 2	1 → 4 → 4 → 4 → 4 → 2 → 2
3	3 → 3 → 3
6 → 1 → 1 → 1 → 2	6 → 6 → 6 → 6 → 6 → 6 → 1 → 1 → 1 → 2 → 2

```
void Stretch(node *list)
{
```