Problem 1: Vectors

a) Vector<eMailMsg> mailVector;

b) void RemoveSpam(Vector<eMailMsg> & v) {
   for (int i = v.size() - 1; i >= 0; i--) {
      eMailMsg mail = v[i];
      if (mail.subject.find("SPAM") == 0) {
         v.removeAt(i);
      }
   }
}

Note that you could work forwards instead of backwards (i.e., loop from 0 to size - 1 instead of the other way around). However, if you did you’d have to make sure to decrement i whenever you removed a message since otherwise you’d skip an index.

c) We use another Vector, of course!

struct eMailMsg {
   Vector<string> to;
   string from;
   string message;
   string subject;
   int date;
   int time;
};

Access to the last element of eMailMsg email would be done by:

string lastAddress = email.to[email.to.size() - 1];

Problem 2: Queues

/**
 * The client version of reverse queue. In order
 * to change the order of elements in the queue,
 * we use an external stack
 */
void ReverseQueue(Queue<int> & queue) {
   Stack<int> stack;
   while (!queue.isEmpty())
   {
      stack.push(queue.dequeue());
   }
   while (!stack.isEmpty())
   {
      queue.enqueue(stack.pop());
   }
}
Problem 3: Using the Scanner and Stack classes

```cpp
#include "stack.h"
#include "scanner.h"

bool ProcessOpenTag(Scanner& scanner, Stack<string>& tagStack) {
    string tag = scanner.nextToken();
    tagStack.push(tag);
    return true;
}

bool ProcessCloseTag(Scanner& scanner, Stack<string>& tagStack) {
    string tag = scanner.nextToken();
    if (!tagStack.isEmpty() && tag == tagStack.pop()) {
        return true;
    } else {
        return false;
    }
}

bool ProcessTag(Scanner& scanner, Stack<string>& tagStack) {
    // read the next token to see if we found an
    // opening or closing tag
    string token = scanner.nextToken();
    if (token == "/") {
        return ProcessCloseTag(scanner, tagStack);
    } else {
        scanner.saveToken(token); // So ProcessOpenTag can use it
        return ProcessOpenTag(scanner, tagStack);
    }
}

bool IsCorrectlyNested(string htmlStr) {
    Scanner scanner;
    scanner.setSpaceOption(Scanner::IgnoreSpaces);
    Stack<string> tagStack;
    scanner.setInput(htmlStr);

    // start by assuming it is balanced
    bool isBalanced = true;
```
while (scanner.hasMoreTokens())
{
    string token = scanner.nextToken();

    if (token == "<")
    {
        if (!ProcessTag(scanner, tagStack))
        {
            isBalanced = false;
            break;
        }

        // get rid of ">") part of tag
        scanner.nextToken();
    }
}

if (!tagStack.isEmpty()) isBalanced = false;

return isBalanced;

Problem 4: Map Warm-up

char MostFrequentCharacter(ifstream &in, int &numOccurrences)
{
    Map<int> charFrequencies;
    numOccurrences = 0;

    int nextChar;
    while((nextChar = in.get()) != EOF)
    {
        // convert it to a string for lookup in the symbol table
        string foundChar = "";
        foundChar += char(nextChar);

        // if we find it, increment the stored value, otherwise
        // enter in a new one
        int frequency = 1;
        if (charFrequencies.containsKey(foundChar))
        {
            frequency = charFrequencies[foundChar] + 1;
            charFrequencies[foundChar] = frequency;
        }

        // now use an iterator to find the most occurring character
        Map<int>::Iterator it = charFrequencies.iterator();
        string maxCharacter = "";
        while (it.hasNext())
        {
            string character = it.next();
            int frequency = charFrequencies[character];

            if (frequency > numOccurrences)
Problem 5: Minesweeper

```cpp
bool LocationOnGrid(int row, int col, Grid<int> & bombCounts)
{
    return row >= 0 && col >= 0 && row < bombCounts.numRows()
        && col < bombCounts.numCols();
}

void MarkBomb(int row, int col, Grid<int> & bombCounts)
{
    for(int bombRow = -1; bombRow <= 1; bombRow++)
    {
        for(int bombCol = -1; bombCol <= 1; bombCol++)
        {
            if(LocationOnGrid(bombRow + row, bombCol + col,
                bombCounts))
                bombCounts(bombRow + row, bombCol + col)++;
        }
    }
}

Grid<int> MakeGridOfCounts(Grid<bool> & bombLocations)
{
    Grid<int> bombCounts(bombLocations.numRows(),
        bombLocations.numCols());
    for(int row = 0; row < bombLocations.numRows(); row++)
    {
        for(int col = 0; col < bombLocations.numCols();col++)
        {
            bombCounts(row, col) = 0;
        }
    }

    for(int row = 0; row < bombLocations.numRows(); row++)
    {
        for(int col = 0; col < bombLocations.numCols();col++)
        {
            if(bombLocations(row, col))
            { 
                MarkBomb(row, col, bombCounts);
            }
        }
    }
    return bombCounts;
}
```
Note that MarkBomb uses two for loops to iterate through the 9 squares it needs to update rather than having a separate case for each square. If it had a separate case for each, this would not only be more messy and less elegant, but it would be more error prone. This is because while writing out 9 different cases, you are much more likely to make an error on one of the lines than if you are only writing out two for loops.