Test 1: CPS 100E

Owen Astrachan and Dee Ramm

October 8, 1996

Name: ____________________________________________ (1 point)

Honor code acknowledgement (signature) _______________________________________

<table>
<thead>
<tr>
<th></th>
<th>value</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem 1</td>
<td>22 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 2</td>
<td>16 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 3</td>
<td>16 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 4</td>
<td>15 pts.</td>
<td></td>
</tr>
<tr>
<td>Extra Credit</td>
<td>8 pts.</td>
<td></td>
</tr>
<tr>
<td>TOTAL:</td>
<td>70 pts.</td>
<td></td>
</tr>
</tbody>
</table>

This test has 11 pages, be sure your test has them all. Do NOT spend too much time on one question — remember that this class lasts 65 minutes.

In writing code you do not need to worry about specifying the proper \#include header files. Assume that all the header files we’ve discussed are included in any code you write.
PROBLEM 1:  *(Dice-man 22 points)*

The declaration for the Dice class and the implementation (both .h and .cc files) are reproduced on page 5.

**Part A:**  *(4 points)*

Briefly describe (you don’t need to write code although you can) how to change the class Dice so that a Dice object defined without specifying the number of sides results in six-sided Dice. For example, Dice cube; makes cube a six-sided Dice. It should still be possible to define Dice tetra(4); after with the changes to the class.
Part B: (4 points)
The function `CountUntilSame` below returns how many times two simulated six-sided Dice are rolled before the same number appears on both dice (using one Dice and rolling it twice correctly simulates two different Dice.)

Modify the code so that it counts how many times three six-sided Dice are rolled before the same number appears on all three dice.

```cpp
int CountUntilSame()
// postcondition: returns # times two 6-side dice are
// rolled until the same number appears on both
{
    Dice d(6); // the die
    int r1,r2; // store the rolls
    int count = 0; // count rolls
    bool allSame = false;

    while (! allSame)
    {
        count++;
        r1 = d.Roll();
        r2 = d.Roll();
        if (r1 == r2)
        {
            allSame = true;
        }
    }

    return count;
}
```

Part C: (4 points)
Write a loop that calls `CountUntilSame` 1000 times and then prints the average number of times three Dice are rolled until the same number appears on all three. Assume that `CountUntilSame` works.

(continued on next page)
Part D: (10 points)
Write a new version of CountUntilSame that has a parameter specifying how many dice are rolled until all show the same value. For example, the call CountUntilSame(8) returns how many rolls are needed before 8 six-sided dice all show the same number.

```c
int CountUntilSame(int numDice)
// postcondition: returns # times numDice 6-side dice are
// rolled until the same number appears on both
{
```

/dice.h

class Dice
{
  public:
    Dice(int sides); // constructor
    int Roll(); // return the random roll
    int NumSides(); // how many sides this die has
    int NumRolls(); // # times this die rolled

  private:
    RandGen myGenerator; // random number generator
    int myRollCount; // # times die rolled
    int mySides; // # sides on die
};

// dice.cc

#include "dice.h"

// implementation of dice class
// written Jan 31, 1994, modified 5/10/94 to use RandGen class

Dice::Dice(int sides)
// postcondition: all private fields initialized
{
  myRollCount = 0;
  mySides = sides;
}

int Dice::Roll()
// postcondition: number of rolls updated
// random 'die' roll returned
{
  myRollCount = myRollCount + 1; // update # of times die rolled
  return myGenerator.RandInt(1,mySides); // in range [1..mySides]
}

int Dice::NumSides()
// postcondition: return # of sides of die
{
  return mySides;
}

int Dice::NumRolls()
// postcondition: return # of times die has been rolled
{
  return myRollCount;
}
PROBLEM 2:  (Friends (16 points))

Part A: (10 points)
A text file stores information about people and each person’s friends. Each line stores a firstname followed by the names of friends as follows.

    owen bob susan fred laura jim claire
    claire craig jim owen
    jim tom fred
    tom laura owen claire

Here tom has three friends (laura, owen, and claire) and jim has two friends (tom and fred).

Write the function below that prints the name of the person who has the most friends. In the example, above owen has the most friends so if the stream input is bound to the data file shown above, the call PrintFriendly(input) prints owen. If more than one person has the same maximal number of friends print any of the the maximally friendly people.

    void PrintFriendly(ifstream & input)
    // precondition: input is open for reading, format as specified in problem
    // postcondition: person with the most friend’s printed
    {

    while (getline(input,line))
    {

    }
Part B: (6 points)

Do NOT write code, but describe how to solve the problem of reading a data file in the format above and finding the best-liked person. The best-liked person is listed as a friend more than any other person. Your description should be sufficient for someone to understand your method and be able to code it.
PROBLEM 3:  (Dominatrix (16 points))

In a double Matrix an entry is dominant if it is the largest element in its row and the largest element in its column (assume all Matrix elements are distinct.) In the diagram of matrix m below the dominant entries are 5.1 at m[1][2], 5.2 at m[3][3], and 5.4 at m[4][1].

```
+---+---+---+---+---+
| 0 | 1 | 2 | 3 | 4 |
+---+---+---+---+---+
| 1.0| 2.0| 5.0| 4.0| 3.0|
| 2.1| 1.1| 5.1| 4.1| 3.1|
| 1.3| 5.3| 2.3| 3.3| 4.3|
| 3.2| 1.2| 2.2| 5.2| 4.2|
| 4.4| 5.4| 3.4| 2.4| 1.4|
```

**Part A:** (10 points)
Write the function Dominators whose header is given below. The function should print the row, column, and value of all dominant entries in the parameter m. For the matrix diagrammed above the output should be:

1,2 = 5.1  
3,3 = 5.2  
4,1 = 5.4

In writing Dominators you may call the functions MaxRowIndex and MaxColIndex whose specifications are provided. **You do NOT need to write MaxRowIndex and MaxColIndex for part A.**

```cpp
int MaxRowIndex(const Vector<double> & v, int size)  
// precondition: # elements in v = size  
// postcondition: returns index of largest element in v

int MaxColIndex(const Matrix<double> & m, int col)  
// precondition: 0 < col < m.Cols() (m has at least col columns)  
// postcondition: returns the index of the largest element in column  
// col of matrix m (where # of rows in m = m.Rows())

void Dominators(const Matrix<double> & m)  
// precondition: m has m.Rows() rows and m.Cols() columns  
// postcondition: row, column, and value of all dominant entries in  
// m are printed
```

(continued on next page)
Part B: (6 points)
Write the function `MaxRowIndex`.

```cpp
int MaxRowIndex(const Vector<double> & v, int size)
// precondition: # elements in v = size
// postcondition: returns index of largest element in v
{
```

```cpp
}
```
PROBLEM 4:  (Rolledex (15 points))

A rolledex is a gadget or program that functions as an address book. A rolledex stores names, phone numbers, and addresses and allows efficient searches for example for all people with the same zip code. The declarations below could be used in a rolledex program.

```cpp
struct Address
{
    string street; // e.g., 103 Erwin Rd
    string zip;   // e.g., 27708
};

struct RolloEntry
{
    string name;
    string phone;
    Address address;
};

class Rollodex
{
    public:
    Rollodex(); // initially empty with room for 100 RolloEntries

    void Add(const RolloEntry &rollo); // add an entry
    void SearchZip(const string &zip, // search for zipcode zip
                    Rollodex &match); // fill match with all matches
    void Print(); // print all entries

    // other member functions

    private:
    Vector<RolloEntry> myList; // store rolledex entries
    int myCount;  // # of entries in myList
};
```

Part A: (5 points)

Write the constructor Rollodex

```cpp
Rollodex::Rollodex()
// postcondition: capacity = 100, # entries = 0
```

(continued on next page)
Part B: (10 points)
Write the member function SearchZip. Provide appropriate pre- and post-conditions. The code below shows a use of the function

```cpp
Rollodex rollo;

// code to put entries into rollo (e.g., read from a file)

string zip;
Rollodex match;
cout << "enter zip to search for: ";
cin >> zip;

rollo.SearchZip(zip, match);
match.Print();
```
Consider a class `StringDice` that behaves like the `Dice` class but which returns strings instead of numbers. For example, the `Roll` function could return a string representing a color of the spectrum: "red", "orange", "yellow", "green", "blue", "indigo", "violet". Alternatively it could return advice (like a magic eight-ball): "all signs are good", "reply hazy ask later", and so on.

Write a class declaration (what would be in a .h file) and code for the class constructor and a member function `Roll` that returns one of a set of strings at random. The constructor for the class should have a vector of strings as a parameter; the vector passes in the strings that will be used to generate a string returned when the `Roll` function is called. The declarations and definition for the `Dice` class earlier on the test may help.