

# Test 2: CPS 006

Owen Astrachan and Susan Rodger

November 15, 1999

Name: \_\_\_\_\_

Login: \_\_\_\_\_

Honor code acknowledgment (signature) \_\_\_\_\_

	value	grade
Problem 1	15 pts.	
Problem 2	30 pts.	
Problem 3	28 pts.	
Extra	10 pts.	
TOTAL:	73 pts.	

This test has 10 pages, be sure your test has them all. Do NOT spend too much time on one question — remember that this class lasts 50 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 :** (*Dang me, dang me, They oughta take a rope and ...* (15 points))

The version of the class `Letters` originally given as part of the Hangman assignment is reproduced below.

```
class Letters
{
public:
    Letters(const string& s);
    bool GuessLetter(const string& letter);
    void Display();

private:

    string myDisplay;
    string myString;
};
```

**Part A (5 points)**

To keep track of missed letters, a student adds the code below to the function `Letters::GuessLetter` to store missed letters in the string variable `misses`. The code doesn't work as intended, a user can guess the same letter many times and the program never prints `already guessed ...`. Explain why the same letter can be guessed more than once and how to fix the problem.

```
bool Letters::GuessLetter(const string& letter)
// post: returns true if letter in secret word, false otherwise
{
    string misses;
    if (misses.find(letter) == string::npos) // not in misses, add it
    {    misses += letter;
    }
    else // in misses so seen it before
    {    cout << "already guessed " << letter << endl;
        return false;
    }
    // more code is here to process letter
```

### Part B (5 points)

A programmer asserts that it's easier to write the code for hangman by creating one class named `Game` that includes all the code from the class `Letters`, the class `WordSource` and the class `Gallows`, because there are fewer concepts to keep track of. In a few sentences describe why using more than one class is a good idea.

### Part C (5 points)

Two programmers are discussing how to add a difficulty level to the hangman game. One says that the class `Letters` object used should be constructed with both the secret word and the number of misses:

```
WordSource ws;  
string s = ws.GetWord();  
Letters letters(s,8);    // allow 8 misses  
// play game here
```

A different programmer claims it's better to track the number of misses in `main` and to determine difficulty level in `main`.

Pick one approach and justify it as better than the other by providing one good reason that it's better.

**PROBLEM 2 :** (*Don't know nothing about Algebra* (30 points))

**Part A (6 points)**

Write the function `Multiply` that multiplies all elements of a `tvector` by a factor. If a `tvector` `vec` is `(1, 3, 6)` originally, then the call `Multiply(vec,4)` should change `vec` to `(4, 12, 24)`.

```
void Multiply(tvector<int>& a, int factor)
// pre: a contains a.size() entries
// post: all entries of a have been multiplied by factor
```

**Part B (7 points)**

Write the function `AddRange` that fills a `tvector` with values between and including `low` and `high`.

For example, the call `AddRange(a,7,12)` would make `tvector a` represent `(7, 8, 9, 10, 11, 12)` if `a` is initially empty. If `a` contained `(8,3)` before the call `AddRange(a,7,12)`, it would contain `(8, 3, 7, 8, 9, 10, 11, 12)` after the call.

```
void AddRange(tvector<int>& a, int low, int high)
// pre: low <= high, a contains a.size() elements
// post: add all values x, low <= x <= high to a
//       adding the values in order to the end of a (calling push_back)
//       a contains a.size() elements
```

*continued next page* →

### Part C (7 points)

A tvector contains positive integers in sorted order, and no number in the tvector is repeated; for example (2, 5, 9, 13, 14). Write the function `GetMissing` that fills the tvector `b` with the positive integers that are missing from `a`, but are between `a[0]` and `a[a.size()-1]`. For example, if `a` is (2, 5, 9, 13, 14) then the call

```
GetMissing(a, b);
```

should make `b` represent (3, 4, 6, 7, 8, 10, 11, 12). In writing `GetMissing` you can call `AddRange` from **Part B**. Assume `AddRange` works as specified.

```
void GetMissing(const tvector<int>& a, tvector<int> & b)
// pre: number of elements in a is a.size(), a is sorted with no duplicates
// post: b contains values missing from a, but between a[0] and a[a.size()-1].
```

### Part D (10 points)

Write the function `RemoveBozos` that removes the names of people who are Bozos from tvector `a` leaving the order of the other elements unchanged. For full credit don't use another tvector in writing the function (the problem is worth 10 points, you can earn a maximum of 7 if you use another tvector). You can call `IsBozo`, do not write it.

```
bool IsBozo(const string& name)
// post: returns true if name is a bozo, false otherwise

void RemoveBozos(tvector<string>& a)
// pre: a contains a.size() entries
// post: all bozos removed from a, order of other elements unchanged,
//      a contains a.size() elements
```

**PROBLEM 3 :** (*www.profrate.com* (28 points))

Students at Duke use a web page to record ratings for each professor they have. Each professor is rated on a scale of 1-10 where 1 is terribly abominable and 10 is supremely wonderful. You're helping to tabulate the results using the struct `ProfRate` below which stores information about one professor's rating and the struct `Student` which stores one student's ratings.

(The functions you write start on the next page.)

```
struct ProfRate
{
    ProfRate()
        : myName(""), myRate(0)
    { }
    ProfRate(const string& name, int rate)
        : myName(name), myRate(rate)
    { }
    string myName;
    int    myRate;
};
```

Each student's ratings are stored in a struct `Student`. Ratings are stored with a student's name so that each student can vote only once. The data field `myRatings` contains ratings inserted using `push_back` so that it contains `myRatings.size()` elements.

```
struct Student
{
    Student() : myName("")
    { }
    Student(const string& name) : myName(name)
    { }
    string myName;
    tvector<ProfRate> myRatings;
};
```

As an example of how these structs are used, the function `Rate` below returns a given professor's (the professor's name is a parameter to the function) rating as determined by one student.

```
double Rate(const Student& s, const string& prof)
// post: returns rating of prof by s, returns 0 if prof not rated
{
    int k;
    int len = s.myRatings.size();
    for(k=0; k < len; k++)
    {   if (s.myRatings[k].myName == prof)
        {   return s.myRatings[k].myRate;
        }
    }
    return 0;
}
```

### Part A (8 points)

Suppose student ratings are stored in a file in the format below: student name on one line followed by several lines with each line containing a rating number followed by the professor's name who achieved that rating. There are as many lines after the student name as there are professor's rated by the student.

```
Joe Student
8 Michael Marxist
7 Nancy Naturalist
2 Henrietta Humanist
```

Complete the function `ReadRatings` below that reads such a file and returns a `Student` object containing the information in the file.

```
Student ReadRatings(const string& filename)
// pre: filename corresponds to a file in the correct format
// post: returns a Student object holding information read from the file
{
    string name;
    ifstream input(filename.c_str());

    getline(input,name); // read first line, get name
    Student s(name);    // construct Student to return

    return s;
}
```

### Part B (12 points)

Write the function `AvgRating` specified below. `AvgRating` returns the average rating of a professor as rated by every student whose information is stored in tvector `a`. For example, if Prof. Smith has a rating of 4, 4, 7, and 9 by four students, the average rating is 6.0. You can call the function `Rate` shown above. If a student doesn't rate a professor, the student doesn't affect the professor's rating.

```
double AvgRating(const tvector<Student>& a, const string& name)
// pre: a contains a.size() entries
// post: returns average rating of professor whose name is 'name'
//       as determined by examining all ratings of all students in a
```

### Part C (8 points)

Describe a method for determining the highest rated Professor at a university, for example by telling how to write `HighestRated` below.

```
string HighestRated(const tvector<Student>& list)
// post: returns most highly rated professor in list
```

Do **NOT** write the code, just write a few sentences telling how you would solve the problem.

**PROBLEM 4 :** (*Extra Credit*)

**Part A (5 points)**

**Do not do this problem until you've done all the other problems on the test**

Suppose you have a user who is really addicted to hangman and plays it 20 or 30 times a week. You want to make sure the user never has to guess the same secret word in the same week. It's ok to guess a word again if it's been seen before but more than one week has passed since it has been the secret word.

Describe at a very high level a method that can solve this problem. Assume that more than one user might be using the same hangman program, but that's ok to save a file of information in the user's home directory (and it's possible for your program to determine the user's home directory).

**PROBLEM 5 :** (*What's the Point? (5 points)*)

**Do not do this problem until you've done all the other problems on the test**

The declaration for the class `point.h` that was used in the random walk examples is reproduced below.

```
struct Point
{
    Point();
    Point(double px, double py);

    string toString() const;
    double distanceFrom(const Point& p) const;
    double x;
    double y;
};
```

The function `MaxDistance` below returns the maximal distance between points in `a`, i.e., if every pair of points is considered, it returns the distance between the two points that are farthest apart. You'll be asked a question about the function after the code.

```

double MaxFrom(const tvector<Point>& a, const Point& p)
// pre: a contains at least one point
// post: returns maximal distance between p and points in a
{
    int k, len = a.size();
    double max = p.distanceFrom(a[0]);
    for(k=1; k < len; k++)
    {   if (p.distanceFrom(a[k]) > max)
        {   max = p.distanceFrom(a[k]);
            }
    }
    return max;
}

double MaxDistance(const tvector<Point>& a)
// pre: a contains a.size() elements
// post: returns maximal distance between points of a
{
    int k, len = a.size();
    double current, max = 0.0;

    for(k=0; k < len; k++)
    {   current = MaxFrom(a, a[k]);
        if (current > max)
        {   max = current;
            }
    }
    return max;
}

```

Executing `MaxDistance` will generate 100 calls of `Point::distanceFrom` for a 10-element vector. It's possible to change the functions `MaxFrom` and `MaxDistance` so that they generate only 55 calls of `Point::distanceFrom`, where  $55 = 1 + 2 + \dots + 9 + 10$ . Describe in a few sentences how to find the maximal distance with a method that will use only 55 calls of `Point::distanceFrom`. Hint: When looking for the maximal distance between the point `a[6]` and other points it's possible to consider only points with indexes 7, 8, and 9. You should describe the algorithm so that it's clear how many distance computations are made.