CSSE 120 Final Exam Topics and Sample Problems

Format: The exam will have two sections:

- **Part 1: Paper-and-Pencil.** Closed-everything, except that you may bring ***TWO*** one-page (back and front) "cheat sheets" with whatever you want on it. Approximately 35 points of 100.
- **Part 2: On-the-computer.** Open-everything, except that you may not communicate with anyone except your professor on this part. Resources you may use include your textbook, your notes, the Internet, homework problems, and the course web site. Approximately 65 points of 100.

Closed book topics: *All* of the closed-book portion of the final exam will be drawn from the following. ***Red arrows mark topics that are especially likely to appear on the final exam.***

For the closed-book portion of the Final Exam, students should be able to:

PYTHON CODE

1. ***Trace short snippets of PYTHON code*** (less than, say, 10 lines or so, per snippet or function) ***and show what gets printed*****.** The code might include anything we discussed, but will attempt to emphasize topics chosen only from the following:

   - **`input`**, **`raw_input`** and **`print`** statements
   - **`range`** expressions
   - Loops that accumulate a:
     - List
     - String
     - Sum
     - Count
     - Max/min

     These loops would typically be a ***for*** loop of any of the following forms (where the first two forms might be over lists, as shown, or over strings):

     - ✓ `for item in list:`
     - ✓ `for k in range(len(list)):`
     - ✓ `for k in range(...):`

   - Nested loops
     - When the inner loop does not depend on the outer loop variable, and also when it does.
   - Function definitions and calls, also method calls
     - Especially with parameters
       - Recall that when you return from a function, the arguments refer to the same object (or number) that they did before the function was called, but the object may have mutated (e.g. if it was a list) during the function call.
     - ***You might see problems very similar to problem 1 and/or problem 3 from the closed-book portion of Exam 2.***
   - Assignment and re-assignment of variables that refer to objects, ***as in problem 4 from the closed-book portion of Exam 2.*** Recall that ***box-and-pointer diagrams*** help you solve such problems.

2. Do the following (and only the following) regarding ***class definitions in Python***:
    - Read the class definition, understanding and being able to explain:
        - What the **`__init__`** constructor is, and when it is called
        - What the ***`self`*** expression means, and why it is not an explicit parameter in the method/constructor call
        - How to identify ***fields*** in the class, noting that they can be used throughout the class and persist as long as the object exists
    - Write, based on the class definition, code that:
        - Constructs an object (calls its constructor)
        - Asks the object to do something (calls a method)
        - References the object's knowledge (references a field of the object)

COMPARING AND CONTRASTING PYTHON AND C

3. ***Compare and contrast Python and C*** by ***writing functions*** or snippets of code ***in BOTH languages*** that solve problems like:
    - Input a number and/or a string, and/or print a number/string/character.
    - Loop using a ***for*** loop with a ***range*** expression (in Python) or a ***for*** loop (in C).
    - Build up (or concatenate to) a list (in Python) or array (in C). The list/array could be built from:
        - A sentinel loop that gets input from the console, or
        - A list (in Python) or array (in C) that already exists.
    - Same as the previous item, but building up a *string*.
    - Counting/summing/max-min/finding/mutating elements in:
        - a list (in Python) or array (in C)
        - a string
    - Defining and calling simple functions that:
        - Print
        - Return a value
        - Mutate a parameter's pointee
4. ***Compare and contrast Python and C*** by ***indicating what is similar and what is different*** in the two languages regarding:
    - Lists (in Python) and arrays (in C)
    - Strings
    - Objects (in Python) and structure instances (in C)
    - Implicit references (in Python) and explicit pointers (in C)
    - Bounds-checking – what are the advantages of doing it, as in Python, or not doing it, as in C?
    - Type declarations – what are the advantages of doing it, as in C, or not doing it, as in Python?
    - Immutability – which of the following are immutable in Python? In C?
        - Numbers
        - Strings
        - Lists (in Python) and arrays (in C)
        - Objects (in Python) and structure instances (in C)

IDEAS COMMON TO PYTHON AND C

5. ***Draw box-and-pointer diagrams*** to indicate what snippets of code involving ***references*** (in Python) or ***pointers*** (in C) are doing.
6. Explain what ***top-down design*** and ***procedural decomposition*** are.
7. Explain what it means to use ***documented stubs*** before coding.

C CODE

8. ***Trace short snippets of C code*** (less than, say, 10 lines or so, per snippet or function) **and show what gets printed.** The code might include anything we discussed, but will attempt to emphasize topics chosen *only* from the following:
    - **`scanf`** and **`printf`** statements
    - **`if-else`** statements
    - Function definitions and calls
        - Especially with parameters, some of which may be pointers
            - Recall that when you return from a function, the arguments refer to the same number, structure instance or pointer (including arrays and strings, which are pointers) that they did before the function was called, but that for pointers (including arrays and strings), the pointee may have mutated during the function call.
        - ***You might see problems very similar to problem 1 and/or problem 3 from the closed-book portion of Exam 2, but translated to C.***
    - Statements involving pointers, pointees and the address-of (&) operator
    - Looping through an array or string, counting/summing/max-min/finding/mutating elements
9. Do the following (but only the following) regarding ***dynamic memory allocation (malloc) in C***:
    - Declare a variable suitable for storing a pointer to memory allocated dynamically (via **`malloc`**)
    - Give that variable an appropriate value (via a call to **`malloc`**)
    - Initialize the allocated space (via a loop, using array notation)
    - Free the space (using a call to **`free`**)
    - Explain why static memory allocation won't allow you to allocate space for an array in a function and refer to that space in the caller
    - Explain why dynamic memory allocation solves this problem
10. Do the following (but only the following) regarding ***structures in C***:
    - Define a structure
    - Declare a variable that is an instance of a structure
    - Reference a field of a variable that is an instance of a structure

Topics you will NOT see on the final exam: *You WILL NOT see the following on the open-book or closed-book portion of the exam:*

- **String and character functions:** I hope that you have learned how to use:
  - the string functions, especially **`strlen`**, **`strncpy`**, **`strncat`**, and **`strncmp`**, and
  - the character functions, especially **`toupper`**, **`tolower`**, **`toascii`** and the various **`isblah`** functions.

  However, you are NOT required to know these functions for the final exam. (You are allowed to use them, but you don't have to do so.) Just be able to loop through a string until its end ('\0'), modifying the string as in the sample problems.
- **Fancy reading and writing from files (getc, fgets, putc, fputs):** I hope that you have gotten some ideas for when to use **`getc`** and **`fgets`** instead of **`fscanf`**, and when to use **`putc`** and **`fputs`** instead of **`fprintf`**. However, the ONLY file-handling functions that you are expected to know for the final exam are **fopen**, **fclose**, **fprintf** and **fscanf**.
- **Pointers to structures:** You saw (briefly) the notation for pointers to structures. However, you will NOT see pointers to structures on the final exam.
- **2D arrays:** You saw the notation for 2D arrays and how to use them. However, you will NOT see 2D arrays on the final exam.

You won't see the following on the open-book portion of the exam, but you may see it on the closed-book portion (in a limited form, see above):

- **Dynamic memory allocation (malloc)**

Open-everything topics, Big Ideas: The sample open-book problems cover 5 problem areas, listed below (and repeated in the sample problems). ***Expect the final exam to have one problem (possibly with subparts) from each of the 5 areas.***

1. A set of problems that lets you demonstrate your understanding of:
    - Input (via `scanf`) and output (via `printf`)
    - Defining and calling functions
    - The difference between a function getting a value via a parameter and the function getting a value via input from the user (`scanf`)
    - The difference between a function printing a value and returning a value
    - Writing and calling functions with pointer parameters that mutate the pointee
2. A problem or problems that let you demonstrate your ability to write **for** loops, including nested loops (loops within loops).
3. A problem or problems that let you demonstrate your understanding of **structures**, in particular, how to:
    - Define a structure
    - Declare an instance of a structure
    - Initialize / reference the fields of that instance
    - Return an instance of a structure from a function, and receive it in the caller
    - Loop through an array of structure instances, examining fields of the array instances and doing something based on that examination, e.g. summing, counting, finding, comparing, max/min, etc.
        - So you need to understand those looping patterns, including how to have a counter for the index of the array, how to increment that counter in a **for** or **while** loop, how to access array elements in the loop, how to do conditional (if-else) statements, etc. To review these patterns, see the next problem area (on arrays).
4. A problem or problems that lets you demonstrate your understanding of arrays (including strings as a special case), in particular, how to:
    - Declare an array
    - Allocate space (statically) for an array
    - Access / loop through / initialize elements of an array
    - Pass an array to a function and receive it as a parameter in the function
    - Perform looping patterns on arrays, including:
        - **for** loop through an array until its end, summing or counting
        - **while** loop through an array/string until a special value is reached (e.g. a space in a string or a positive number in an array of numbers)
        - **while** loop through a string until its end (`'\0'`) is reached

- **while** loop (perhaps loop-and-a-half-pattern) that gets input from the console until a sentinel value is reached, or from a file until end-of-file (see next problem for the latter), filling the array with the input values
- Nested loops through arrays/strings (see examples below)
- Loop in which you deal with an array element at more than one index each time through the loop (see examples below)

5. A problem or problems that lets you demonstrate your understanding of reading from and writing to files.
    - Just how to open the file (with **`fopen`**), close the file (with **`fclose`**), read from the file (with **`fscanf`**) and write to the file (with **`fprintf`**).

Sample closed-book problems:

PYTHON CODE

1. What gets printed by the following **Python** code?

```
def silly(x):
    print "start silly"
    print x
    goofy(x - 1)
    funny(x + 2)
    print "end silly"

def funny(y):
    print "start funny"
    print y
    goofy(y * 2)
    print "end funny"

def goofy(x):
    print "start goofy"
    print x
    print "end goofy"

silly(3)
```

Output:

2. What gets printed by the following **Python** code?

```
z = range(10, 25, 5)
print z
```

Output:

3. What gets printed by the following **Python** code?

```
dogs = ["fido", "ruff"]
foo = []
k = 1
for dog in dogs:
    foo.append(k)
    foo.append(dog)
    k = k + 1
print k
print foo
```

Output:

4. What gets printed by the following **Python** code?

```
numbers = [10, 15, 3, 17]
foo = ""
for k in range(len(numbers)):
    foo = foo + str(k) + " " + str(numbers[k]) + " "
print foo
```

Output:

5. Below is a portion of a Python program to simulate a simple alarm clock.
   Read the code and then answer parts a-d.

```
class Clock:
    def __init__(self, hours, minutes):
        """ Assume a 24-hour clock, so 0<=hours<=23, 0<=minutes<=59"""
        self.hours = hours
        self.minutes = minutes

    def tick(self):
        self.minutes += 1
        if self.minutes == 60:
            self.minutes = 0
            self.hours += 1

        if self.hours == 24:
            self.hours == 0

        if self.hours==self.alarmHours and self.minutes==self.alarmMinutes:
            print "ALARM: ", self.hours, self.minutes

    def setAlarm(self, alarmHours, alarmMinutes):
        self.alarmHours = alarmHours
        self.alarmMins = alarmMinutes
```

   a. What are the names of two of the fields in the Clock class?

   b. Write a line of code that creates a Clock object called **myClock** with the time set to 5:59.

   c. Write a line of code to then cause **`myClock`**'s time to advance to the next minute.

   d. I set **myClock**'s alarm to 8:30 by making the call

      ```
      myClock.setAlarm(8, 30)
      ```

      In the definition of **setAlarm**, 3 formal parameters are shown, but I call it with only 2 actual arguments. Explain.

COMPARING AND CONTRASTING PYTHON AND C

6. Consider the following problem: A function takes a list (in Python) or an array (in C) of integers. The function returns the sum of the integers in the list/array.
    - ***Write a function in Python*** that solves the above problem.
    - ***Write a function in C*** that solves the above problem (you'll need to add a second parameter to the function).
7. Consider the following problem: A function takes a list (in Python) or an array (in C) of strings. The function concatenates all the strings into one long string. (So if the list/array contained three elements:

    "Hello there."

    "How are you?"

    "I am fine."

    then the function would produce the string "Hello there.How are you?I am fine.")
    - ***Write a function in Python*** that solves the above problem. The Python function should *return* the string that the function forms.
    - ***Write a function in C*** that solves the above problem. The C function will need two additional parameters – one for the length of the array and another that will hold the string that the function forms. The C function should put the concatenated string that it forms into that last parameter. (Aside: What assumption do you need to make about that last parameter?)
8. Compare and contrast Python and C by indicating what is similar and what is different in the two languages regarding:
    - Lists (in Python) and arrays (in C)
    - Strings
    - Objects (in Python) and structure instances (in C)
    - Implicit references (in Python) and explicit pointers (in C)
    - Bounds-checking – what are the advantages of doing it, as in Python, or not doing it, as in C?
    - Type declarations – what are the advantages of doing it, as in C, or not doing it, as in Python?
    - Immutability – which of these are immutable in Python? In C?
        - Numbers
        - Strings
        - Lists (in Python) and arrays (in C)
        - Objects (in Python) and structure instances (in C)

IDEAS COMMON TO PYTHON AND C

9. ***Draw box-and-pointer diagrams*** to indicate what the following snippets of code are doing. Also show what is output.

a. Python code:

```
def foo(x, y, z):
    x = x + z[1]
    y = [1, 2, 3]
    z[0] = z[1]
    z[1] = z[2]
    print x, y, z

a = 4
b = [5, 10, 15]
c = [8, 30, 60]
foo(a, b, c)
print a, b, c
```

Box and pointer diagrams (you can just cross out things to show how they change as the code executes):

Output:

b. C code:

```
void foo(int x, int* y, int z[]) {
    x = x + z[1];
    y = &x;
    z[0] = z[1];
    z[1] = z[2];
    printf("%d %d %d %d\n", x, *y, z[0], z[1]);
}

int main() {
    int a = 4;
    int b = 100;
    int* c = &a;
    int d[] = {8, 30, 60};
    foo(b, c, d);
    printf("%d %d %d %d %d\n", a, b, *c, d[0], d[1]);
    return EXIT_SUCCESS;
}
```

Box and pointer diagrams (you can just cross out things to show how they change as the code executes):

Output:

10. ***Draw box-and-pointer diagrams*** to indicate what the following snippets of code are doing. Also show what is output.

a. Python code:

```
circleA = Circle(Point(25, 25), 10)
circleB = Circle(Point(50, 50), 20)
circleC = Circle(Point(75, 75), 30)

circleC = circleB
circleB = circleA

circleA.move(100, 0)
circleB.move(200, 0)

print circleA.getCenter().getX()
print circleB.getCenter().getX()
print circleC.getCenter().getX()
```

Box and pointer diagrams (you can just cross out things to show how they change as the code executes):

Output:

b. C code:

```
int main() {
    int a = 10;
    int b = 20;
    int c = 30;

    int* p1 = &a;
    int* p2 = &b;
    int* p3 = &c;

    printf("%d %d %d\n", *p1, *p2, *p3);

    p3 = p2;
    p2 = p1;

    *p1 = *p1 + 100;
    *p2 = *p2 + 100;

    printf("%d %d %d\n", *p1, *p2, *p3);

    return EXIT_SUCCESS;
}
```

Output:

Box and pointer diagrams (you can just cross out things to show how they change as the code executes):

C CODE

11. Write the following in C (all are good sample problems):

    a. Suppose that you want to dynamically allocate 200 Fish objects, where Fish is a structure that you have defined. Write a statement that declares a variable suitable for a pointer to the 200 Fish objects (in the ***next*** part of this problem you will allocate those 200 Fish objects).

    b. Continuing the previous problem, give your variable an appropriate value (via a call to **`malloc`**).

    c. Suppose that **Course** is a structure that you have defined, and that Course has a field called **numberOfStudents**. Suppose that **csse120** is an instance of the Course structure. Write an appropriate declaration for **csse120**.

    d. Continuing the previous problem, write a statement that sets the **numberOfStudents** field of **csse120** to *23*.

    e. Suppose that your program already contains a function called **uglyNumber** whose specification and prototype are as follows:

```
// Returns the kth Ugly Number.  (Note: you do not need to
// know what an Ugly Number is to solve this problem.)
int uglyNumber(int k);
```

    Write a function called **printUglies** that takes integers ***m*** and ***n*** and prints the *m*th Ugly Number through the *n*th Ugly Number, inclusive.

Sample open-book problems (all in C): The sample open-book problems cover 5 problem areas, listed below. ***Expect the open-book portion of the final exam to have one problem (possibly with subparts) from each of the 5 areas.***

1. A set of problems that lets you demonstrate your understanding of:
   - Input (via `scanf`) and output (via `printf`)
   - Defining and calling functions
   - The difference between a function getting a value via a parameter and the function getting a value via input from the user (`scanf`)
   - The difference between a function printing a value and returning a value
   - Writing and calling functions with pointer parameters that mutate the pointee

   Here are some examples of the above. For each of the following prototypes:
   - ***implement the function*** and then
   - ***write code in main that tests*** your implementation.

   a. `void printDoubleInput()`

   Prompts for and gets from the console an int *x*, doubles *x*'s value, and prints an appropriate message to the console. For example:

   ```
   printDoubleInput();
   ```

   would print

   ```
   Enter an integer:
   ```

   and if the user then entered (say) 8, it would print

   ```
   x = 8, double x = 16
   ```

   b. **`void printDoubleValue(int x)`**

   Receives an int *x*, doubles *x*'s value, and prints an appropriate message to the console. For example:

   ```
   printDoubleValue(3);
   ```

   would print

   ```
   x = 3, double x = 6
   ```

   c. **`int returnDoubleValue(int x)`**

   Receives an int *x* and returns the value that is 2 times *x*

   d. **`void mutateFloat(float* x)`**

   Receives a pointer *p* to a floating point value. Doubles the value of *p*'s pointee.

By the way, the functions **swap** and **minAndMax** in **24-CPointers** are other good examples of this last problem.

2. A problem or problems that let you demonstrate your ability to write **for** loops, including nested loops (loops within loops).

   The problems in **23-CForLoops** are good examples of such problems – review your solutions to those problems. Here is another example:

   Write a function

   ```
   void printHollowRightTriangle (int size)
   ```

   that takes a single parameter, which is the size (width and height) of the right triangle. The function should print a right triangle of that size, made of asterisks, but "hollow". You may assume that the size is a positive number greater than 3. For example,

   ```
   printHollowRightTriangle(6)
   ```

   should print

   ```
   *
   **
   * *
   *  *
   *   *
   ******
   ```

   I'll comment that such problems often have the following features:

   - There is a loop within a loop, with the outer loop going from row to row and the inner loop doing each row.
   - For "triangular" shapes like the above, the inner loop range often depends on the outer loop variable.
   - There are often special rows that must be done with their own loop (instead of being part of the loop within a loop). For example, can you see which row in the above problem is special in this way?
   - The row itself may have special parts. For example, in the above problem, what must be printed in a loop for each row? What must be printed before that loop? After that loop?

3. A problem or problems that let you demonstrate your understanding of **structures**, in particular, how to:
    - Define a structure
    - Declare an instance of a structure
    - Initialize / reference the fields of that instance
    - Return an instance of a structure from a function, and receive it in the caller
    - Loop through an array of structure instances, examining fields of the array instances and doing something based on that examination, e.g. summing, counting, finding, comparing, max/min, etc.
        - So you need to understand those looping patterns, including how to have a counter for the index of the array, how to increment that counter in a **for** or **while** loop, how to access array elements in the loop, how to do conditional (if-else) statements, etc. To review these patterns, see the next problem (on arrays).

The problems in **25-Structures1** are good examples of such problems – review your solutions to those problems. Here is another example: (continues on next page)

Consider a **TestScore** structure defined as follows:

```
typedef struct {
    char studentUsername[9];
    int studentID;
    float test1Score;
    float test2Score;
    float finalExamScore;
} TestScore;
```

Implement the following functions:

```
// Returns a TestScore constructed from the information
// sent as parameters.

TestScore makeTestScore(char name[], int id,
                float test1, float test2, float finalExam) {
     // Implement this function.
}
```

```
// Takes an array of TestScore structure instances and the
// length of that array.  Returns the studentID of the
// student who improves the most from test1 to test2.
// If no student improves, returns -1.
// If there is tie for most-improvement, you can return
// the studentID of any of the tied students.

int mostImmprovement(TestScore scores[], int length) {
     // Implement this function.
}
```

Note: If you want a harder version of this problem, have the ***mostImprovement*** function take a second array that is an array of int's, and put into that array the studentID's of all the students who are most-improved from test1 from test2. (So only one studentID goes into the array if there are no ties, but more than one goes into it if there are ties for most-improved.)

4. A problem or problems that lets you demonstrate your understanding of arrays (including strings as a special case), in particular, how to:
    - Declare an array
    - Allocate space (statically) for an array
    - Access / loop through / initialize elements of an array
    - Pass an array to a function and receive it as a parameter in the function
    - Perform looping patterns on arrays, including:
        - **for** loop through an array until its end, summing or counting
        - **while** loop through an array/string until a special value is reached (e.g. a space in a string or a positive number in an array of numbers)
        - **while** loop through a string until its end (`'\0'`) is reached
        - **while** loop (perhaps loop-and-a-half-pattern) that gets input from the console until a sentinel value is reached, or from a file until end-of-file (see next problem for the latter), filling the array with the input values
        - Nested loops through arrays/strings (see **Type 2** example below)
        - Loop in which you deal with an array element at more than one index each time through the loop (see **Type 3** example below)

    Many problems are possible here as examples. Some good ones from your homework are:
    - All the problems in the **26-Arrays1** project from Homework 26
    - All the problems in the **HousePrices** project from Homework 26.
    - Many of the problems in **27-CharactersAndStrings** project from Homework 27, especially the **stringLength**, **firstWord**, and **reverse** functions that you implemented.

    Here are several more sample problems: (continues on the next page)

Implement the following functions:

TYPE A: Problems that require a single loop through the array.

```
// Returns the sum of the numbers in the given array.
// The array has the given length.

double sumArray(double numbers[], int length) {
      // Implement this function.
}
```

```
// Returns true if the given array contains a number less than
// the given number, else returns false.
// The array has the given length.

int sumArray(double numbers[], int length, double compareToMe) {
      // Implement this function.
}
```

TYPE B: Problems that require a loop within a loop through the array.

One example is to sort the array using **selection sort**, as you did in the **HousePrices** project from Homework 26. (See that homework for a description of selection sort.)

Here is another example: (In doing this example, first figure out how to solve the problem and, as part of that problem-solving, why you need a loop within a loop; only then turn to implementing your idea in C.) (continues on next page)

```
// Returns true (1) if the given array contains TWO DISTINCT
// elements whose sum equals the given goal, else returns
// false (0).  The array has the given length.
// For example, suppose    int a[] = {5, 12, 3, 8, 12, 11, 2, 6};
// Then  pairSum(a, 8, 14);  returns 1 (True) because 12 + 2 = 14
// and   pairSum(a, 8, 24);  returns 1 (True) because 12 + 12= 24
// and there are two 12's in a.
// But  pairSum(a, 8, 4);  returns 0 (False) because no pair of
// DISTINCT numbers in the array have 4 as their sum.

int pairSum(int numbers[], int length, int goal) {
      // Implement this function.
}
```

TYPE C: Problems in which you loop through the array but deal with an array element at more than one index each time through the loop.

One superb example of this type of problem is the **reverse** function from the **27-CharactersAndStrings** project in Homework 27. In one way of solving this problem, you loop through the first half of the array and *swap* the kth element with its mate at the other end of the array, each time through the loop.

Here is another example whose simplest solution fits this type of pattern:

```
// Returns true (1) if the given array contains an element
// that equals the sum of the next two elements in the array,
// else returns false (0).  The array has the given length.
// For example, suppose:
//      int a[] = {20, 10, 4, 8, 3, 5, 20};
//      int b[] = {20, 10, 4, 8, 3, 5, 20};
// Then  nextTwoSum(a, 7);  returns 1 (True) because 8 = 3 + 5
// but   nextTwoSum(b, 7);  returns 0 (False)

int nextTwoSum(int numbers[], int length) {
      // Implement this function.
}
```

5. A problem or problems that lets you demonstrate your understanding of reading from and writing to files.

Implement the following functions:

```
// Opens the file whose name is given, for writing.
// Prints an error message and returns -1 if the file cannot
// be opened for writing.  Writes the given string
// into the file (overwriting whatever was there, or
// creating the file if it did not previously exist.)
// Closes the file and then returns 0.

int toFile(char* string) {
      // Implement this function.
}
```

```
// Opens the file whose name is given, for reading.
// Prints an error message and returns -1 if the file cannot
// be opened for reading.  Assume each line of the file
// contains a word, an integer, and a floating point number,
// separated by spaces.  Read the data in the file and:
//   -- Print the words, in the order they appear in the file.
//   -- Put the integers into the first of the given arrays,
//      in the order they appear in the file.
//   -- Put the floating-point numbers into the second of
//      the given arrays, in the order they appear in the file.
//      (Assume that both arrays are big enough.)
// When done reading, close the file
// and return the number of lines in the file.

int fromFile(char* filename, int numbers1[], double numbers2[]) {
      // Implement this function.
}
```