CSE 303 Midterm Exam

October 29, 2008

Name ________Sample Solution_________

The exam is closed book, except that you may have a single page of hand-written notes for reference.

If you don't remember the details of how a specific function or command works (parameter lists, whether an error is indicated by a non-zero return code or null pointer, etc.), write down the assumptions you need to make and as long as they're reasonable you're ok.

If you have questions during the exam, raise your hand and someone will come to you. Don't leave your seat.

Please wait to turn the page until everyone has their exam and you have been told to begin.

Advice: The solutions to many of the problems are quite short. Don't be alarmed if there is a lot more room on the page than you actually need for your answer.

More gratuitous advice: Be sure to get to all the questions. If you find you are spending a lot of time on a question, move on and try other ones, then come back to the question that was taking the time.

1	/ 12
2	/ 6
3	/ 16
4	/ 17
5	/ 16
6	/ 17
7	/ 16
Total	/ 100

Question 1. (12 points) Write regular expressions that could be used with grep to search the word file we were using in class to find words with the following patterns. Each line of the file contains a single word, with no leading or trailing whitespace.

(a) All words that begin with the three letters “bor” and end with the three letters “ing”.

```
^bor.*ing$
```

(the ^ and $ are needed to avoid picking up words that have the regular expression as part of the word, but not at the beginning and end.)

(b) All words that contain 3 or more occurrences of the same vowel (aeiou). This means 3 or more occurrences of a, or of e, or of i, etc., not 3 or more vowels in any combination.

```
\([aeiou]\).*\1.*\1
```

Question 2. (6 points) Write a Unix command line (not a shell script) that will search the word file `words.txt` described in problem 1 (i.e., one word per line, no leading or trailing whitespace), and print the number of words that contain the string “science” in them.

The solution we originally had in mind was

```
grep science words.txt | wc -l
```

but a few people also had this, which, of course, also received full credit:

```
grep -c science words.txt
```

It turns out that in this particular problem `wc -w` would also work, because there is only 1 word per line, and we gave full credit for that. But in general you should use `wc -l` to count lines.

Question 3. (16 points) The input to this problem consists of a file with information about students and their exam scores. Each line contains information about one student: their exam score, their student number, and their last name. The fields are separated by ':' characters. For example:

```
82:0639864:Smith
97:0892736:Jones
6:9835467:Perkins
97:0367453:Kobashi
```

Write a (very short) shell script that will read files in this format and copy the names and scores to standard output sorted by name and formatted as follows (using the example data above):

```
Jones: 97
Kobashi: 97
Perkins: 6
Smith: 82
```

You may assume that the input file exists, is readable, and has the right format. The file name is given as an argument to the shell script (i.e., the script should read the named file and not read from stdin).

Hints: sed, sort

```
#!/bin/bash
sed 's/\(.*\):.*:\(.*\)/\2: \1/' "$1" | sort
```

Some notes:

1) **Many solutions had patterns that matched `[0-9]*` or something similar for the scores and/or the id numbers. That's fine, but not strictly needed in this particular case, since the ':' characters in the regular expression anchor the search.**

2) **In a robust shell script, the argument `$1` should be quoted as above. We didn't deduct points if you didn't do that on the test.**

3) **We also didn't deduct points for scripts that wrote the `sed` output to a file named, for example, `temp`, then sorted `temp` and later removed it. In a real script this would be a bad idea, since it would trash any file named `temp` that already existed in the directory.**

Question 4. (17 points) (The (almost) obligatory twisty pointer problem.)

The following program compiles and executes without errors. What does it print?

```
#include <stdio.h>

void mumble(int **p, int * q) {
  **p = 10;
  *p = q;
  printf("mumble: **p = %d, *q = %d\n", **p, *q);
}

int main() {
  int k = 3;
  int n = 7;
  int *p = &k;
  mumble(&p, &n);
  printf("main: k = %d, n = %d, *p = %d\n", k, n, *p);
  return 0;
}
```

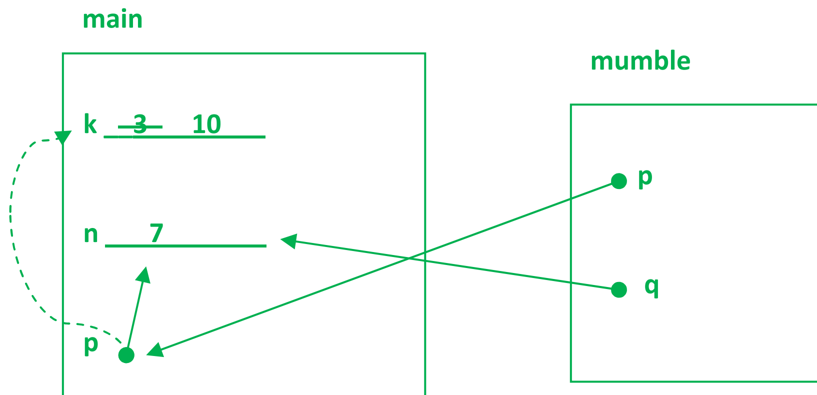
Advice: Drawing diagrams can be very helpful in solving the problem and at least as helpful to the grader in figuring out how to assign partial credit in the unlikely event that your answer isn't exactly right. ☺

Output:

```
mumble: **p = 7, *q = 7
main: k = 10, n = 7, *p = 7
```

Diagrams and other scratch work:

(Diagrams were not required, but you *really* should learn to draw them if you haven't yet. This is the situation right before the return from `mumble`.)



Question 5. (16 points) The following function is, as explained in the comments, supposed to reverse the order of the characters in the string that is its argument.

```
#include <string.h>
#include <stdlib.h>

/* reverse the order of the characters in the string s. */

void reverseString(char * str) {

  char* i, *j;     // pointers to the beginning and end of the string

  int len = strlen(str);

  if (len == 0) {

    return;

  }

  i = str;

  j = str + len-1;

  while (i < j);

  {

    char tmp = *i;

    *i = *j;

    *j = tmp;

    i++;

    j--;

  }

}
```

Missing * before j, not a pointer without it

Parameter is s in some places, str in others; use one or the other

Need == for equality test, not = (assignment)

Off by 1 error – need to start at character before the \0 to avoid swapping the \0 to the front.

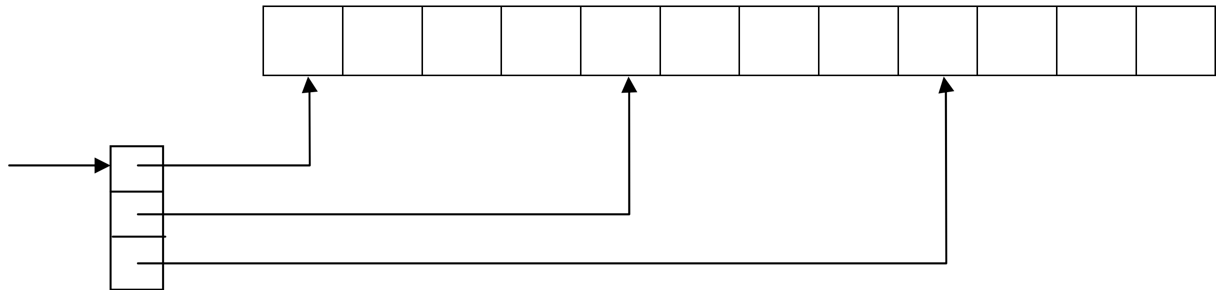
Extra ';' after while condition – infinite loop; delete it

Mixing characters and pointers in the swap; add pointer dereferences as needed

Unfortunately, there are bugs in this function. Indicate the problems in the above code and describe how to fix them by writing in corrections, adding code if needed, or crossing out incorrect code and showing how to fix it.

For full credit you should fix the existing code and not rewrite it to do the required work in a different way.

Question 6. (17 points) 2-dimensional arrays are common in many scientific and engineering problems. There are several ways to represent them in C, but a common and flexible scheme is to allocate a single 1-dimensional array with room for all of the data elements, one row after another, and a second array of pointers, where each element in the pointer array is initialized to point to the first element in the corresponding logical row of the data. The total number of elements in the data array is the product of the number of rows and number of columns. For example, an array with 3 logical rows, each of which contains 4 columns of data, would look like this.



Complete the definition of function allocate2D below so that it dynamically allocates this data structure and returns a pointer to the pointer array. The data elements themselves should have type double and do not need to be initialized. The pointer array needs to have an appropriate type and its elements need to be initialized to point to the logical rows in the data.

```
/* Allocate a new "2D" array with nr rows and nc columns, as   */
/* described above, and return a pointer to the pointer array. */
double** allocate2D(int nr, int nc) {

  double *d;      /* pointer to data array */
  double **p;     /* pointer to pointer array */
  double *temp;
  int k;

  /* allocate arrays */
  d = (double *)malloc(nr*nc*sizeof(double));
  p = (double**)malloc(nr*sizeof(double*));

  /* initialize pointer array */
  temp = d;
  for (k = 0; k < nr; k++) {
    p[k] = temp;
    temp += nc;
  }

  return p;

}
```

```
/* another possible loop */
for (k = 0; k < nr; k++) {
  p[k] = &d[k*nc];
}
```

Question 7. (16 points) For this problem, complete the following C program so that it reads the file whose name is given on the command line and prints out the number of lines in the file and the file name (i.e, it should produce the same output as “wc –l <filename>” – an integer with the number of lines in the file followed by the file name). The program can be written as a single main function, although you may define additional functions if you wish. You may assume that no input line has more than MAX_LINE_SIZE characters, including the trailing ‘\0’. You do not need to check for errors – assume that the input file exists and can be read successfully.

Hints: fopen(char* filename, char* mode); fgets(char *buffer, int max_len, FILE *stream);

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_LINE_SIZE 1000     /* max # characters per line */

/* Print the number of lines in and name of the input file */
int main(int argc, char** argv) {

  int nlines;                  /* number of lines read so far */
  char line[MAX_LINE_SIZE];    /* space for input data */
  FILE * f;                    /* input file */

  f = fopen(argv[1], "r");
  nlines = 0;
  while (fgets(line, MAX_LINE_SIZE, f)) {
    nlines++;
  }

  printf("%d %s\n", nlines, argv[1]);

  return 0;

}
```

Several solutions dynamically allocated the input line on the heap with `malloc`. That’s fine, except in that case, a `free` was needed to return the allocated data. Technically, it would also be a good idea to use `fclose` to release the file after reading it, but we didn’t expect that or deduct points if it was not done.