# CSE 303 Midterm Exam

**February 11, 2009**

**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.

Please wait to turn the page until everyone has their exam and you have been told to begin. If you have questions during the exam, raise your hand and someone will come to you. Don't leave your seat.

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. Also, there may be more than one correct answer to a problem – any of them will be fine as a solution.

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 | / 15 |
| 2 | / 12 |
| 3 | / 17 |
| 4 | / 14 |
| 5 | / 14 |
| 6 | / 12 |
| 7 | / 16 |
| Total | / 100 |

**Question 1.** (15 points) Write regular expressions that could be used with grep to to find lines in a file that contain strings with the following patterns.

(a) All lines that begin with an upper-case letter (‘A’ through ‘Z’).

```
^[A-Z]
```

(b) All lines containing at least two adjacent digit characters (i.e., 17, 01, 42, 33, 00, etc.).

```
[0-9][0-9]
```

(c) All lines containing six-character sequences that form a palindrome (i.e., six characters that are the same forward and backwards like “pullup” or “123321” or “ah!!ha”).

```
\(.\)\(.\)\(.\)\3\2\1
```

**Question 2.** (12 points) (a) Write a Unix command line (not a shell script) that will print the number of files and directories in the current directory. It should not count files and directories whose names begin with a dot ("."). The command should just print a single number.

```
ls | wc -l
```

**There are a couple of similar commands that are almost correct, but have tiny bugs that cost a point.**

```
ls | wc -w          (counts each word separately in files with names containing blanks)

ls -l | wc -l       (there's an extra "total" line included in the count)
```

(b) Suppose we have a long text file named `story.txt`. Write a Unix command line (not a shell script) that will display the first few lines of `story.txt` in the terminal window (i.e., at least the first 5 or 6 lines of the file, but no more than will fit in the window.) The output does not need to completely fill the terminal window, but it must contain the first few lines.

```
head story.txt or cat story.txt | head
```

**The commands `more` and `less` also work equally well for this problem.**

**Question 3.** (17 points) Give the code for a bash script that deletes files containing more than a given number of bytes. Suppose that the script arguments are named `sz, f1, f2, f3, ..., fn`. Argument `sz` is the size (number of bytes) and arguments `f1` through `fn` are the file names. The script should behave as follows:

- If no arguments are given (i.e., `sz` is missing,) the script should write an error message to `stderr` and exit with a non-zero exit status.
- If at least the first argument (`sz`) is present, the script should process each of the file arguments `f1` through `fn` in turn. For each file argument,
    - If the file does not exist or is not a regular file, then ignore that argument and continue processing with the next file argument. Do not print any messages.
    - If the file does exist and is a regular file (i.e., not a directory or some other kind of special file), then if it contains more than `sz` bytes it should be deleted.

    If there are no file arguments (i.e., only `sz` is present), the script should terminate without doing anything further.

Hint: `wc –c`, bash test condition `-f`.

```
#!/bin/bash
if [ $# -lt 1 ]
then
     echo "missing size parameter" >&2
     exit 1
fi
sz=$1
shift
while [ $# -gt 0 ]
do
     if [ -f "$1" ]
     then
          if [[ $sz –lt `cat "$1" | wc –c` ]]
          then
               rm "$1"
          fi
     fi
     shift
done
```

**There are several variations that also work. The inner if condition can be written as (( sz < … )).**

**There are other variations that almost work and we didn't deduct points. These include using –lt inside ((…)) or < inside of [[…]], or using […] instead of [[…]]. There's also a subtle bug with wc –c "$1", since the output from wc includes the filename as well as the size. When wc reads from stdin it doesn't include a filename in its output.**

**Question 4.** (14 points) A small business keeps their employee address book in a plain text file. Each line in the file contains information about one employee. The information on each line has three fields separated by exclamation marks (`!`). The fields contain the employee name, office number (building and room), and phone number. For example:

```
Paul Allen!Sieg 134!5-3425
Susan Sieg!CSE 403!555-1212
Wilma Flintstone!Sieg 505!3-5367
Fred Flintstone!CSE 555!3-7452
Alan Turning!CSE 705!44-0-1908-640404
```

Write a short shell script that will copy the names and phone numbers of people in a particular building to `stdout`. The script has two arguments: the first is the name of the text file containing the information in the format described above; the second is the building name.

For example, if the first argument to the script is name of a file containing the above data, and the second argument is the string `Sieg`, the script should produce the following output:

```
Paul Allen 5-3425
Wilma Flintstone 3-5367
```

These are the only two entries in the original file with offices in Sieg. (Note that Susan Sieg's office is in the CSE building, so her name and phone number do not appear in the output.)

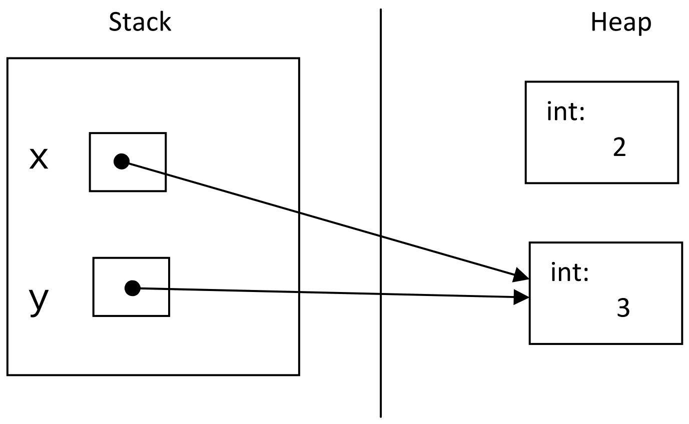
Your script should print the message "wrong number of arguments" and exit with a non-zero exit status if it is not given exactly two arguments. You do not need to check for additional errors: assume that the input file exists and can be read, etc., if the correct number of arguments is given. If the building (argument 2) does not exactly match a building name in the file (argument 1) then the script should produce no output.

Hints: sed, grep; a pipeline might be useful.

```
#!/bin/bash
if [ $# -ne 2 ]
then
     echo wrong number of arguments
     exit 1
fi
grep \!.*"$2".*\! "$1" | sed 's/\(.*\)!.*!\(.*\)/\1 \2/'
```

**As before there are many variations that work fine, and some small errors that we ignored. For example, \! needs to be escaped to keep from having the shell interpret it as a history command and we let that go. But you can't put ! inside single quotes with $2, or the $2 won't get expanded and that is an error. We also ignored it if people put $2 right next to the preceding !, although that isn't quite as robust as the solution here. Another, more clever sed command that some people had, was `sed 's/!.*!/ /'`. That works because sed only needs to delete part of the line but not rearrange the order in this problem. A bug that several people had was to pipe the output from sed to echo. That doesn't make sense because echo doesn't read from stdin.**

**Question 5.** (14 points) Consider the following drawing



Complete the definition of function `puzzle` below so that it creates the above pointer structure, including allocating both `int`s on the heap and arranging the pointers as shown. (And yes, this diagram does display a memory leak.)

```
void puzzle {
   int * x;
   int * y;

   x = malloc(sizeof(int));
   y = malloc(sizeof(int));

   *x = 2;
   *y = 3;

   x = y;

}
```

**A cast to (int*) isn't technically needed in front of the calls to malloc, but it's fine to put it in, and it is better style. Without the cast, C silently converts the pointer value from malloc to the type of the variable.**

**Question 6.** (12 points) What does the following program print when it is executed? It does compile and execute successfully without errors.

Suggestion: draw a diagram of what happens during execution. That should help you answer the question, and could help us award partial credit if your answer isn't quite right.

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

int main(int argc, char *argv[]) {
  char s[25] = "ants are in my pants";
  s[2] = s[3];
  strcpy(s+4, "error");
  s[6] = '\0';
  s[3] = 'w';
  printf("s: %s\n", s);
  return 0;
}
```

Output produced:

```
s : answer
```

**Question 7.** (16 points) Complete the definition of function `contains`, below, so it returns true (1) if a particular string appears somewhere in a linked list of strings, and false (0) if not.

The nodes in the linked list are defined by the following struct:

```
struct node {               /* one node in the linked list: */
    char *s;                /*   string belonging to this node */
    struct node * next;     /*   next list node, or NULL if none */
};
```

In other words, each node contains a pointer to the string associated with that node, and a second pointer to the next node in the list, or NULL if this is the last node in the list. The list might be initially empty (i.e., the list pointer might be NULL). The string must match one of the strings in the list exactly for the function to return true.

Hint: `strcmp(char *s1,char *s2)` returns a negative integer if string s1 is less than s2, a positive integer if s1 is greater than s2, and 0 if they are the same.

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

/* return 1 if string str appears in list lst or 0 if not */
int contains(char *str, struct node *lst) {

   struct node *p = lst;

   while (p != NULL) {
      if (strcmp(p->s, str) == 0) {
         return 1;
      }
      p = p->next;
   }

   return 0;

}
```

**The extra local variable p isn't needed, but your instructor prefers it as a matter of style, instead of modifying the parameter lst.**

**Many people abbreviated the loop and if conditions to things like `while (p != 0)` or `while(p)` or `if (!strcmp(p->s, str))`. Those idioms are common enough in C that it's a good idea to understand them; whether it makes things clearer or is better style is something of a personal preference.**