The University of Chicago
Department of Computer Science

CMSC 16200 – Honors Introduction to Computer Science 2
Winter Quarter 2013
Lab #4

Instructions

This lab consists of two exercises. The first is to be done individually. The second consists of 3 parts: the first two to be completed as a group, and the last one individually.

At the beginning of the lab session, we will choose groups of 3. It is up to the members of each group whether to work on the group part of Exercise 2 during the lab session, or in a separate meeting.

Your individual submission should contain:

- Your (individual) code for Exercise 1.
- Your group's files (DOCUMENTATION.txt, **`goodcat.c`**) for the first two parts of Exercise 2. All members of a group should submit (copies of) the same two files, and the group members should be listed at the beginning of DOCUMENTATION.txt.
- Your (individual) code for the third part of Exercise 2.
- (A) Makefile(s) to compile your code and your group's code (see instructions on last page). Different group members do not need to use the same Makefile to compile the group's code (**`goodcat.c`**).

Exercise 1

You will write a program that allows a user to specify an array of potentially infinite size.

1. The program will repeatedly ask the user the following two questions:

```
Enter a position:
Enter a value:
```

The first time this pair of questions is asked (let's assume the user specifies pos=5, value=7), the program will need to dynamically allocate enough memory for an array with six positions, and assign value 7 to position 5.

For all subsequent entries, the program will behave the following way:

- If the current array can accommodate the requested assignment, then no change is necessary to the array. Simply perform the assignment. For example, if the user specifies pos=3 and value=17, we can do the assignment because our array has six positions.
- If the current array is too small to accommodate the requested assignment, then you need to create a *new* array with enough positions, copy all the data from the old

array into the new array, and then do the assignment. For example, if the user specifies pos=25 and value=123, we need to create a new array of size 26, copy all the data from the old array, and then perform the assignment.

You may assume that the user will only specify non-negative integers for both the position and the value.

After each assignment, your program should ask the user if he/she wants to specify another assignment (y/n question, no need to validate this input).

2. When the user is done assigning values, you must print the contents of the array, as follows:

```
array[0] = 3
array[1] = 37
array[2] = [No value assigned]
array[3] = [No value assigned]
array[4] = 1990
array[5] = [No value assigned]
array[6] = [No value assigned]
array[7] = [No value assigned]
array[8] = 42
```

Note: You will have to keep track of which positions have a value assigned to them. You are *not* allowed to have a second array with the sole purpose of keeping track of what positions have, or do not have, a value. (Clue: The fact that you can assume that *values* are non-negative integers is relevant for this.)

3. Finally, you will ask the user *once* to specify a position, and will print the number at that position. For example, using the above array:

```
What position do you wish to access? 1
array[1] is 37

What position do you wish to access? 5
No value at that position

What position do you wish to access? 50
Not a valid position
```

You must write this part of the exercise by implementing the following function:

```
int getValue(??? array, int numElements, int pos, ??? value);
```

You will need to decide what the parameter types should be for `array` and `value`.

Parameters:

- `array`: The array specified by the user.
- `numElements`: The number of elements in the array
- `pos`: Array position to access
- `value`: Output parameter where the value is to be deposited.

Return value:

- 0: If the specified position is valid.
- 1: If the specified position is valid, but there is no value in that position.
- 2: If the specified position is not valid (out of bounds)

Exercise 2

The following program is a naïve implementation of the UNIX standard command `cat` (you can find the source code in the lab wiki as `badcat.c`).

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

#define SUCCESS  0
#define E_PARAM  1

int main(int argc, char **argv)
{
    int i, numread;
    FILE *in;
    char buf[100];

    if(argc==0)
    {
        fprintf(stderr, "ERROR: Not enough parameters.\n");
        fprintf(stderr, "Syntax: %s [file1] [file2] ... [fileN]\n", argv[0]);
        exit(E_PARAM);
    }

    for(i=1;i<argc;i++)
    {
        in = fopen(argv[i], "rt");
        if(in==NULL)
            fprintf(stderr, "\n%s: %s: No such file or directory\n", argv[0], argv[i]);
        else while(!feof(in))
        {
            numread=fscanf(in, "%s", buf);
            if(numread>0 && numread != EOF)
                    fprintf(stdout, buf);
        }
    }
    exit(SUCCESS);
}
```

However, this implementation contains several errors, the most important of which are:

- On any input file, the output has been stripped of all whitespace (which is not the desired outcome of `cat`)
- The program will segfault on certain files (a sample `segfault.txt` file is available in the lab wiki).

Furthermore, there is at least one other error related with file I/O which does not cause any visible effect but is, nonetheless, Bad Karma.

You are asked to do the following:

- As a group, find and describe the errors in this program. In the case of the segfault error, you should explain *why* the provided `segfault.txt` file is making the program crash. Write your solution to this part of the exercise in a file called `DOCUMENTATION.txt`.
- As a group, modify the provided source code, correcting the errors you identified. The corrected file should be called **`goodcat.c`**
- The provided implementation is not only wrong, it is also rather inefficient. *Individually*, come up with a more efficient implementation. (Hint: The inefficiency of the above program stems from the fact that we are using high-level stream functions like `fscanf`.) You will need to use lower-level I/O calls. Your solution should be called **`goodcat_indiv.c`**

Using **make**

When submitting your code, you should also include a Makefile to compile your code. At the very least, you should include a separate Makefile for each exercise, so that the code can be built like this:

```
$ make -f Makefile.ex1

$ make -f Makefile.ex2

$ make -f Makefile.ex2indiv
```

You can also write your Makefile using different styles (e.g. a single Makefile with separate targets for each exercise, and a default target that will build all exercises). However, in that case you must include a README file with your submission with instructions on how to compile your code. *Makefiles should work on a CS Linux machine*. Failure to include a Makefile will negatively impact your grade.