CS 352 (Fall 16): System Programming and UNIX

Project #4

Dynamic Memory Allocation

due at 9pm, Wed 28 Sep 2016

1 Overview

This project has three small programs. The first does not require any dynamic memory allocation; you can do it all with fixed-size arrays (like the programs from Project 3). It involves searching through a long string for instances of a small string.

However, the 2nd and 3rd programs both require that you use `malloc()` or `calloc()` to allocate dynamic memory. This is because you do not know, before your program runs, how much memory will be required.

In a real-world program, you often have to **resize** your memory as your program runs - and we'll get to that later. But for this project, we'll simplify things: we will tell you, at the start of the testcase, how much memory is required. You simply need to allocate it (just once), and then use it throughout your program.

(Looking for the "Standard Requirements" and "Grading" sections? I've moved them to the end.)

2 `turnin`

Turn in the following directory structure using the assignment name `cs352_f16_proj04`:

```
substrings/
    substrings.c
scanList/
    scanList.c
scheduling/
    scheduling.c
```

REMEMBER: Submit only these three directories! Do not place them inside another directory!

3 Program - Find Substrings

In this program, you will read in a single long string (we'll call it the "Master String"), and then compare multiple other strings to it, checking to see if any are substrings of the first string. In this program (unlike previous programs) you will read entire lines and save them as your strings - instead of using `scanf()` to read whitespace-delimited strings.

Name your file `substrings.c`

3.1 Input

The input to this program is simply a set of lines of text. It should always have at least one line (the "Master String"), but it might not have any lines after that.

We will guarantee that every line of input is no longer than 120 characters (not counting the newline); this means that if you want to use `fgets()`, you may; simply ensure that you have space for a worst case string. (What is the worst case string if the line has 120 characters? Read the man page to find out!)

The first line of a testcase should never be empty; if it is, your program must report an error and terminate. However, after that, any and all lines can be empty; you will simply skip over these lines.

3.2 Reading the Input

You probably will want to use either `fgets()` or `getline()` to read the lines of input. You can assume that the line input will not have more than 120 chars, but your program should not have memory errors if a larger line is put in. For the subsequent strings, do not count the newline as part of the search string. For example, if the Master String is:

```
The dog wags his tail.
```

and the line

```
dog
```

is entered after, the result should be "4". In other words, `dog` is a substring even though there is no newline after `dog` in the Master String.

3.3 Behavior

Your program will first read the Master String. If you cannot read any data (because of EOF), or if the line is entirely empty, then report an error and terminate your program immediately.

Otherwise, read other lines from the input. Skip over any empty lines that you find; for each non-empty line, scan through the Master String to see if any subset of it matches the line you just read.

If your program finds a match, print out the index of the first character of the match. (Don't look to see if there are any more matches; ignore them even if they might exist.) If your program cannot find **any** match in the Master String, print out -1.

3.4 Error Conditions

A blank initial line or no lines input should cause the program to print an error message to stderr and exit. After the Master String is read, additional blank lines should NOT be treated as errors. They should just be ignored. (Note, only lines that are empty are "blank". A line that contains spaces should be treated as a valid string.)

3.5 Restrictions

The student must write their own function for checking if one string is the substring of another. You are NOT allowed to use built in string functions like `strstr()` to do this for you. You can use the string fuctions `strlen()`, `strcpy()`, `strdup()` if you want. Ask if you want to use a different string function.

3.6 EXAMPLE

Suppose that the input to your program is this:

```
The quick brown fox jumps over the lazy dog.
foo
quick
Quick
n fox
.
asdf
The q
```

your output should be:

```
-1
4
-1
14
43
-1
0
```

4 Program - Scan the List

In this program, you will read in an array of integers. You will then read in a series of **other integers**; for each integer you read, you will count how many times it showed up in the original array.

Name your file `scanList.c`

4.1 Input

The input is a series of integers. The first integer (which gives the count of the number of integers in the comparison-array) must be non-negative; the rest of the integers can take any value.

4.2 Behavior

First, read a single integer; allocate an array of integers of this size.

Next, read in the array. Each value in this set must be an integer, but could be negative, positive, or zero. After you have read this entire array, print it out like this:

```
Comparison array (10 elements): 123 0 -27 14 0 13 123 100 14 6
```

Finally, read the rest of the input; it should be entirely integers. For each integer count how many times this value shows up in the array. Print it out like this:

```
Number: 123 count=2
```

4.3 Error Conditions

As always, check the return code from all operations that could fail (such as `malloc()/calloc()` and terminate your program with an error message if it fails.

- If the input contains anything other than integers, report an error when you read it, and terminate your program.
- If the first value on the input is an integer but is negative, then report an error and terminate your program.
- Finally, if you are unable to read the full set of integers required (for the initial array), report an error and terminate your program.

Otherwise, your program should report no errors. In particular, it is **perfectly OK** if the input has no values to search for at all.

5 Program - Scheduling

In this program, you will look for overlaps in the schedule for several people, and build a report about what times they might or might not have available. This will require that you allocate an array of integers, to keep information about each of the slots.

Name your file `scheduling.c`

5.1 Input

The input begins with a positive integer, which gives the number of slots in the schedule. It is then followed by any number of pairs of numbers. Each pair represents a range of slots; for instance the pair `3 6` means slots `3,4,5,6`. Each range represents an existing appointment held by some person.

The numbers will be separated by whitespace, so you are encouraged to use `scanf()` with the `"%d"` format specifier to read them.

There can be any number of ranges.

5.2 Behavior

First, your program should read the count of slots, and allocate an array of `int` of that size (using `malloc()` or `calloc()`). Initialize all fields to zero (if you use `calloc()`, then `calloc()` will do this for you).

Next, loop through the input, reading pairs of integers. For each pair, do some simple checking:

- Verify that both numbers are valid slot numbers (non-negative, and less than the number of slots)
- Verify that the first is less than or equal to the second.

If you could not read two integers, report an error and terminate the program. If you you read two integers but their values were invalid, report an error, skip over this range, and continue reading from input.

For each range that is valid, iterate through the slots and increment each slot in the range. In this way, each integer in the array will keep a count of how many ranges covered that slot.

When you have read the last range, dump out the array. Print the number of slots, a colon and a space, and then a space-separated list of all of the slot values, like this:

```
5: 10 3 3 4 5
```

Then scan through the slots, and report the following information:

- The minimum and maximum values found in each slot; also report the list of **slots** which have the min and max.
- The average value found in all of the slots (use a floating point value; use the format specifier `"%.2f"` to force `printf()` to print out two digits to the right of the decimal).

The output should look like this:

```
minimum: 3
minimum-slots: 1 2
maximum: 10
maximum-slots: 0
average: 5.00
```

5.3 EXAMPLE

Suppose that your input is as follows:

```
5
0 4
0 4
0 4
3 4
4 4
1 1
1 1
```

Your output should be:

```
5: 3 5 3 4 5
minimum: 3
minimum-slots: 0 2
maximum: 5
maximum-slots: 1 4
average: 4.00
```

5.4 Error Conditions

If the input does not begin with a positive integer, report an error and terminate the program immediately. If the input begins with a positive integer, allocate an array of integers of that size; if `malloc()` fails, report an error and terminate the program immediately.

If you cannot read a range (either because there is a single integer, and then EOF, or because there is a non-integer input), report an error and terminate the program immediately.

If you read two integers for a range - but the integers are invalid - then report an error, but continue running the program.

5.4.1 Exit Status

If the program runs with no errors, its exit status should be 0. If there were one or more errors (even if the program kept running), its exit status should be 1.

5.5 Hints

- Use a cast to get a floating-point result from division. (Remember, an integer divided by an integer is another integer - rounding down.)
- Use `printf()` with the format specifier `"%.2f"` to print nice floating point values.

6 Standard Requirements

- Your C code should adhere to the coding standards for this class:
`http://www.cs.arizona.edu/classes/cs352/fall16/DOCS/coding-standards.html`

- Your programs should indicate whether or not they executed without any problems via their **exit status** - that is, the value returned by `main()`. If you return 0, that means "Normal, no problems." Any other value means that an error occurred.

  In this class, we will always use the value 1 to indicate error; however, outside this class, many programs use many different exit status values - to indicate differnt types of errors.

  In `bash`, you can check the exit status of any command (including your programs) by typing `echo $?` immediately after the program runs (don't run **any** commands in-between).

6.1 Special Note: `scanf()` and `%s`

Several programs in this project (and in later projects) will require you to read strings from `stdin` using `scanf()`. **This can be dangerous, if you read more data than your buffer allows** - since it is possible to read right off the end of the array, and overwrite other memory.

To solve this, `scanf()` allows you to limit the number of characters that you read with the `%s` specifier. **You must always use this feature.** Remember that `scanf()` will also write out a null terminator - so this number **must** be less than the size of your buffer, like this:

```
char buf[128];
int rc = scanf("%127s", buf);
```

7 Grading

We have provided compiled versions of these programs for you; you should download them and run them (on Lectura) for comparison. The programs can be copied from `/home/cs352/fall16/Assignments/proj04/` on Lectura. (The UNIX commands you've learned, such as `ls, cd, cp` will all be useful here.)

We have also provided our grading script. (We'll do this for the first couple of assignments, in order to help you understand the requirements - but we will stop doing this as the programs get more complex!) You can copy it from `/home/cs352/fall16/Assignments/proj04/grade_proj04` on Lectura, or from the web.

Finally, we have provided a few example inputs to test your program with. (We will use additional ones when we grade your program.) **You should write**

additional testcases - try to be creative, and exercise your program in new ways.

NOTE: Since this project has multiple programs, it needs testcases for each one. The testcases are named `test_<progName>_*`. In order for the grading script to see your new testcases, please name them according to this standard.

7.1 Exact Output

In each Project this semester, most of your grade will come from automatic testing of the code. You must match `stdout` **EXACTLY**, byte for byte. Common mistakes include:

- Extra or missing spaces
- Extra or missing blank lines
- Misspelled words
- Different capitalization

Your code must also give the same return value (0 or 1) as the example executable in every case.

`stderr` will be handled a little more loosely. In each testcase, we will check to see if the standard executable reported **some** error message to `stderr` or not. If it did, then your program must as well; if not, then your program must not print anything, either. However, we will not be comparing the exact error messages.

NOTE: Each testcase either passes, or fails entirely. We do not give partial credit for any testcase - although you may, of course, pass some testcases but not others.

7.2 Running the Grading Script

To run the grading script, arrange your files like this, and then run `./grade_proj04`

```
grade_proj04

example_substrings
example_scanList
example_scheduling

test_substrings_*
test_scanList_*
test_scheduling_*

substrings/
    substrings.c
```

```
scanList/
    scanList.c
scheduling/
    scheduling.c
```

The grading script does a number of things:

- Confirms that we have an example executable for each program.
- Confirms that each of the required files has the proper name, in a directory with the proper name. If not, you lose all points for that program.
- Compiles your code. If your code doesn't compile at all, then you lose all points for that program. If it compiles but has warnings, then you will get a heavy deduction.
- Runs your code against all of the testcases. In each case, it runs both your code, and also the example executable. It checks to make sure that both have the same return code - and then also checks to make sure that the outputs match. If not, then it will give you a zero for that testcase.

  (If you have no testcases for some program, then the script will give you a zero for that program.)

At the end, the grading script reports a score; this score is determined by the number of testcases that you passed - modified for any deductions.

7.3 Hand Grading

In addition, each program will be looked at by a human, to check for things which we can't automatically check, like:

- Some of the details of the spec
- Good comments
- Good indentation
- Reasonable variable names

7.4 Point Distribution

In this project, your code will be graded by a script, with your score determined by how many testcases you pass for each program. After that score is calculated, a number of penalties may be applied.

7.4.1 Weight of each program:

If any program compiles and runs, but warnings were produced by the compiler, you will lose half your score for that program.

- 30% - `substrings`
- 30% - `scanList`
- 40% - `scheduling`

7.4.2 Possible Penalties

- -10% - Poor style, indentation, variable names
- -10% - No file header, or missing header comments for any function (other than `main()`)
- -20% - Any use of `scanf("%s")` without limiting the number of characters read