CS 352 (Fall 16): System Programming and UNIX

Project #3

Strings are Arrays of Characters

due at 9pm, Wed 21 Sep 2016

1 Overview

This project has three small programs. Each one deals with strings (which in C, are null-terminated arrays of characters) and with the basic characters themselves.

1.1 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).

2 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);
```

3 `turnin`

Turn in the following directory structure using the assignment name `cs352_f16_proj03`:

```
dumpStrings/
    dumpStrings.c
removeSome/
    removeSome.c
stringSearch/
    stringSearch.c
```

4 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/proj03/` 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/proj03/grade_proj03` 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.

4.1 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.

4.2 Running the Grading Script

To run the grading script, arrange your files like this, and then run `./grade_proj03`

```
grade_proj03

example_dumpStrings
example_removeSome
example_stringSearch

test_dumpStrings_*
test_removeSome_*
test_stringSearch_*

dumpStrings/
    dumpStrings.c
removeSome/
    removeSome.c
stringSearch/
    stringSearch.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.

4.3 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

4.4 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.

4.4.1 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% - `dumpStrings`
- 30% - `removeSome`
- 40% - `stringSearch`

4.4.2 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
- -10% - In `dumpStrings`, if you do not print out **both** a count that you came up with - and also the value returned by `strlen()`
- -10% - In `stringSearch`, if you fail to use `strcmp()` to do the "Mat" comparison, or if you use it for either the "Rev" or "Dup" comparisons.

5 Program - Dump the Strings

In this program, you will dump out each string that you read from `stdin`. You will read each string with `scanf()`, into a buffer. You will then iterate through the string, and print out each character: you will print out the character itself, and its decimal and hexadecimal values. At the end of each string, you will report some simple metadata about the string.

Name your file `dumpStrings.c`

5.1 Input

The input is a sequence of strings. These strings may contain any mix of text, numerals, or symbols, provided that:

- No string is longer than 31 bytes
- All of the characters in each string are printable symbols (no strange control characters!)

You are not required to verify either of these requirements; the testcases will enforce this for you. **However,** you must design your program such that it can read the worst-case string (which is 31 characters, not counting the null terminator).

5.2 Behavior

For each string in the input, loop over the characters in that string. For each character, print out the index into the array, the character itself (surrounded by single quotes), its decimal value, and its hexadecimal value. For instance, if the character at index 7 was a lowercase a, then you would print:

```
index=7 char='a' dec=97 hex=0x61
```

NOTE: `printf()` can print uppercase or lowercase hexadecimal digits. For this project, you just print lowercase.

Stop printing when you hit the null terminator for the string; do **not** print out a line for the null terminator itself.

After you hit the end of the string, print out a line which summarizes what you found. Report both a length that you counted in your loop - and also report the value returned by the C standard library function `strlen()`:

```
count=13 strlen=13
```

End each string dump with a single blank line, including the last. (If there is nothing at all to read from `stdin`, then print out nothing at all.)

WARNING: We will check your code to make sure that you print out both something you counted, and also `strlen`. Do both, or you will lose points. But they should always print out the same thing!

5.3 Error Conditions

There are no error conditions to handle in this program. Your program should never print to stderr, and must always return an exit status of 0.

5.4 Example Output

Suppose that your input was as follows:

```
asdf
jkl
qwertyuiop[]1234567890
```

The appropriate output from your program would be (it's hard to see, but this includes a blank line at the very end):

```
index=0 char='a' dec=97 hex=0x61
index=1 char='s' dec=115 hex=0x73
index=2 char='d' dec=100 hex=0x64
index=3 char='f' dec=102 hex=0x66
count=4 strlen=4

index=0 char='j' dec=106 hex=0x6a
index=1 char='k' dec=107 hex=0x6b
index=2 char='l' dec=108 hex=0x6c
count=3 strlen=3

index=0 char='q' dec=113 hex=0x71
index=1 char='w' dec=119 hex=0x77
index=2 char='e' dec=101 hex=0x65
index=3 char='r' dec=114 hex=0x72
index=4 char='t' dec=116 hex=0x74
index=5 char='y' dec=121 hex=0x79
index=6 char='u' dec=117 hex=0x75
index=7 char='i' dec=105 hex=0x69
index=8 char='o' dec=111 hex=0x6f
index=9 char='p' dec=112 hex=0x70
index=10 char='[' dec=91 hex=0x5b
index=11 char=']' dec=93 hex=0x5d
index=12 char='1' dec=49 hex=0x31
index=13 char='2' dec=50 hex=0x32
index=14 char='3' dec=51 hex=0x33
index=15 char='4' dec=52 hex=0x34
index=16 char='5' dec=53 hex=0x35
index=17 char='6' dec=54 hex=0x36
index=18 char='7' dec=55 hex=0x37
```

```
index=19 char='8' dec=56 hex=0x38
index=20 char='9' dec=57 hex=0x39
index=21 char='0' dec=48 hex=0x30
count=22 strlen=22
```

6 Program - Remove Some

In this program, you will read strings from the `stdin`. For each string you will first check to see if it is entirely made up of letters and/or decimal digits. If it is entirely made up of letters, you will print out the letters, but with all of the vowels removed. If it is entirely made up of decimal digits, then you will print out the digits **in reverse order**, but with all of the even-numbered digits removed. (That is, remove the 0's, 2's, 4's, 6', and 8's, wherever they are found in the string). If you read a word which is neither all letters nor all decimal digits, you will print that word out (with an error message) to `stderr`.

Name your file `removeSome.c`

6.1 Input

The input is a sequence of strings, separated by whitespace. Strings may be up to 31 characters long. You do not need to validate these assumptions - although your code must be able to handle the longest, worst-case string (including the null terminator).

However, you must verify the **contents** of these strings. Strings ought to be either all letters or all decimal digits; if any string is neither of these, you must print out an error message.

While no string can mix letters and numbers, a given testcase can include both strings that have letters, and other strings which have numbers.

Strings that have letters may include any mix of lowercase and uppercase letters.

6.2 Behavior

Read each string from `stdin`. For each string, first check to see if it is either of the two types expected (all letters or all digits). If not, report an error to `stderr` and move on. If so, then print out the string to `stdout` (modified as required), one string per line.

As noted in the overview above, when you print out a string of letters, you must remove all vowels (both lowercase and uppercase vowels). When you print out a string of digits, you must skip every even-valued digit, while printing them in reverse order.

REMINDER: In English, there are 5 vowels: a,e,i,o,u. Your program should remove both uppercase and lowercase vowels.

6.3 Hint

Check the man page for `isdigit()` and `tolower()`. In each man page, note that there are also **other** functions which might also be useful for this program!

6.3.1 Hint

There are several ways to edit the string. A good, simple solution would be to use the `%c` format specifier of `printf()`, to print out one character at a time. That is the algorithm we use in our solution to this problem.

However, there are other options. One is to allocate a **pair** of buffers, and to copy digits from one buffer to another. If you are careful to copy over a null terminator, this works - and then you can simply call `printf()` with `%s`.

In theory, it would also be possible to modify the string in-place, inside a single buffer. While that works, it will be **hard to get right,** and so we don't really recommend that you try it.

6.4 Error Conditions

If you find any word on the input which is neither all letters, nor all numbers, then you should print out an error message to `stderr`. However, the program should not die; it must continue to read additional strings.

The exit status of your program should be:

- 0 if there were no errors on the input
- 1 if there were errors on the input (even if the program was able to keep running)

All error messages must be printed to `stderr`.

7 Program - String Search

In this program, you will first read a single string from the input. You will then read additional strings. Print to output any string which **exactly** matches the original word, which matches it in reverse order, or which **exactly** matches the previous string read. In each case, we have provided a message that you must print, along with the string, to explain why you printed it.

Name your file `stringSearch.c`

7.1 Input

The input is at least one string (the search string). There could be zero, one, or many words that follow it. The strings will be no longer than 31 characters, and they will be separated by whitespace. The strings may contain any printable ASCII character.

You are not required to confirm that the input follows this description - except that you must confirm that at least one string (the search string) exists. Otherwise, you may simply assume that the testcase follows the specification above.

Your program **must** be able to handle the worst-case string (that is, 31 bytes long, not counting the null terminator).

7.2 Behavior

Your program should first read in the search string; save it to a buffer. Then enter a loop where you read other strings frrom the input, until you hit EOF. For each string you read, print it out if:

- Print it out if it exactly matches the search string, in order. Print out the message `"Mat:"`, a single space, then the string, then a newline.
- Print it out if it matches the search string, but in reverse order. Print out the message `"Rev:"`, a single space, then the string, then a newline.
- Print it out if it is an exact duplicate of the previous string read. Print out the message `"Dup:"`, a single space, then the string, then a newline.

(It is possible that some lines might be hits for multiple reasons - for instance, if the search string was a palindrome, then each hit would be both a "Mat" and a "Rev." Or, a "Dup" might also be a "Rev." In those cases, print out a line for each type of match: first "Mat," then "Rev," then "Dup.")

At the end of the program, print out a single-line summary, describing the number of each type of match, like this:

```
Totals: strings=10 : m=1 r=2 d=0
```

where `strings` is the number of strings read (not including the very first), and `m,r,d` are the number of each type of match. (Note that some or all of those numbers might be zero!)

7.2.1 About Duplicate Strings

When checking for duplicate strings, you compare the string you just read with `scanf()` to the previous one that you read. This means that you will need two buffers - and the ability to copy from one to another.

Perform this check on the second string in the input. For instance, if your input is this:

```
foo foo bar oof baz baz foo
```

your output should be:

```
Mat: foo
Dup: foo
Rev: oof
Dup: baz
Mat: foo
Totals: strings=6 : m=2 r=1 d=2
```

The first line happens because the second word of input matches the first. The second is because the first and second are duplicates (both "foo"). The third line is because "oof" is "foo", reversed. The fourth line is because the string "baz" shows up twice in a row. The fourth is because the string "foo" shows up again at the end of the input.

7.3 Hint

Check out the man page for `strcpy()`.

7.4 Special Requirements

In the Real World, C programmers use the standard library function `strcmp()` to compare two strings. We want you to get used to `strcmp()`, but we also want you to get experience looping over arrays. For that reason, we have a strange requirement for this program:

- For the "Mat" comparison, you **must** use `strcmp()` to compare the strings.
- For the "Rev" and "Dup" comparisons, you **must not** use `strcmp()`; instead, you must write your own loop. (If you want to encapsulate the logic in a function, that's a great idea - but the function must not call `strcmp()` !)

7.5 Error Conditions

If the input stream has no strings in it at all, print out an error message to `stderr` and terminate the program with an exit status of 1. Otherwise, your program should not ever print to `stderr`, and should always return 0.