Project Strings + Performance

Due: October 25, 2016

Table of Contents

1 Overview

This assignment was previously two separate projects — Strings and Performance — that for logistical reasons, were combined into a single project with two parts.

You should approach each of the two parts as its own project, independent of the other part.

In the Strings portion of this project, you will implement a subset of C's string manipulation library.

In the Performance portion, you will optimize the performance of a function using techniques you learned in class.

2 Install the Stencil

To get started, run the following command in a terminal:

```
cs033_install stringsperf
```

This will install the stencil code and a copy of this handout in your *course/cs033/stringsperf* directory.

The only files you should edit are *strings/strings.c* and *perf/poly.c*.

3 Part 1: Strings

Dory is having trouble remembering everything about her family. However, sudden flashbacks bring back random strings of letters which she writes on paper. To sort through this immense collection of seemingly senseless words, the little fish needs to manipulate and understand strings using a rather witty approach.

String manipulation is an important concept in computer science, and it is something that comes up very often in systems programming. Most programming languages have a string library that relieves programmers from writing their own string operations for every program. The C Standard Library has some excellent string manipulation facilities, but we know that Dory deserves a set of functions that she can really call her own!

Let's help Dory understand her memories and find out about her past!

3.1 Assignment

For this part of the project, you will implement a subset of C's standard library string functions, which you will use in upcoming Shell assignments to tokenize and search strings.

You should implement the following functions, all of which are documented extensively in the `strings.c` stencil file. As a hint, you may need to reuse earlier functions in later functions, so it might help to write these functions in order.

Your implementations should be efficient, but do not optimize your code at the expense of readability. (You will get a chance to write highly optimized code in Performance!) Please comment your code and explain any complicated logic.

Here are the functions you should write, along with example uses of each:

- `size_t strlen(const char *s);`
  `strlen` computes the length of a string, exluding the terminating null byte.

```
size_t len = strlen("CS 33 Rocks!"); // len = 12
```

- `size_t strspn(const char *s, const char *accept);`
  `strspn` computes the number of bytes in the largest prefix of `s` that contains only characters from `accept`.

```
char *s = "we all love strings";
char *accept = "we alove";
size_t span = strspn(s, accept); // span = 12
```

- `size_t strcspn(const char *s, const char *reject);`
  `strcspn` computes the number of bytes in the largest prefix of `s` that contains only characters not in `reject`.

```
char *s = "coming up with example strings gets hard";
char *reject = "breakingthe4thwall";
size_t span = strcspn(s, reject); // span = 3
```

- `int strncmp(const char *s1, const char *s2, size_t n);`
  `strncmp` is similar to the behavior of `strcmp`, which you do not have to implement for this assignment. `strcmp` compares two strings and returns a number less than, equal to, or greater than 0, if `s1` is found to be lexicographically less than, equal to, or greater than `s2`.

  `strncmp` is similar, except that it only compares the first (at most) `n` bytes of `s1` and `s2`.

```
int result = strncmp("same", "samexyz", 4); // result = 0
```

- `char *strncpy(char *dest, const char *src, size_t n);`
  `strncpy` is similar to the behavior of `strcpy`, which you do not have to implement for this assignment. `strcpy` copies the contents of `src` into `dest`

  `strncpy` is similar, except that it only copies the first (at most) `n` bytes from `src` into `dest`.

```
char dest[] = "abcdefg";
strncpy(dest, "xyzt", 3); // dest now becomes "xyzdefg"
```

- `char *strstr(const char *haystack, const char *needle);`
  `strstr` finds the first occurrance of the string `needle` in the string `haystack`.

```
char *needle = "33";
char *haystack = "---32---33---33---";
char *location = strstr(haystack, needle); // location = "33---33---"
```

- `char *strtok(char *str, const char *delim);`
  `strtok` returns a pointer to the first segment of `str` that does not contain any characters in `delim`. `strtok` should return non-empty token strings or NULL if there are no more tokens in `str`. Note that `strtok` is stateful; to extract a sequence of tokens from the same string, you must make multiple calls by specifying the corpus string in the first call and NULL each time after.
  Hint: `strtok` makes no promise to leave `str` untouched. You may overwrite characters with null terminators if you'd like.
  Hint: The `static` keyword may be useful when writing `strtok`.

```
char *delim = "-";
// string is a char * that equals "---I---Love---33---"
char *token1 = strtok(string, delim); // token1 = "I"
char *token2 = strtok(NULL, delim); // token2 = "Love"
```

Remember that in C, you can think of the type `char *` as any of three things: a pointer to a character, a string, or an array of characters. These are all the same! Because of this, you can index into a string like an array.

3.2 Allowed Library Functions and Resources

Read the man pages for more information about each of the string functions if you're confused by the explanation, or come to hours and discuss the functions in more detail. All functions that you'll need to use are defined in `strings.c`. **There are no external functions allowed.**

3.3 Testing

We have included a few tests in the main function of the `tests.c` file. These tests will test the expected functionality of this string library. While they do cover basic functionality, we encourage you to write more of your own tests so that you can be sure that your functions work correctly before using them in subsequent projects.

Please do not modify the tests that are already in the file, as this will only make it harder for you to confirm that everything is working. Also, we will test your code with no compiler optimizations, so do not use a compiler flag to improve your times.

To build the test executable, run:

```
make
```

To test your work against the entire test suite, run:

```
./run_tests [number of repetitions] all
```

To test your work against a specific test or list of tests, run:

```
./run_tests [number of repetitions] <test name(s)>
```

We will be testing your code's efficiency with one million repetitions. If you do not specify a number when running the tests, it will default to one million.

Here are some reference times you should shoot for. Each time is in milliseconds.

	Baseline	Optimized	System
strlen	824ms	357ms	255ms
strspn	2s 339ms	608ms	571ms
strcspn	2s 365ms	624ms	580ms
strncmp	1s 956ms	700ms	679ms
strncpy	273ms	53ms	41ms
strstr	487ms	148ms	32ms
strtok	371ms	221ms	177ms

We are looking for times that run faster than the baseline. See if you can reach the optimized times! The system column shows the times of the actual system functions.

4 Part 2: Performance

Nemo, in an effort to become the fastest fish in school, has decided to speed things up. He's going to start by calculating polynomials faster and better.

4.1 Assignment

The purpose of this assignment is to utilize various techniques to optimize the speed of C code. You will be optimizing the performance of a function `poly` and explaining the techniques that you used.

A baseline implementation of `poly` is provided. Your task is to identify and implement ways to improve the function's performance. Please note that to implement a performant `poly` function you may need to restrict the range of degrees you can pass in, but you must at a minimum support degrees greater than or equal to 15 (you don't need to worry about the result overflowing when the number of degrees is sufficiently high).

4.2 `poly`

```
double poly(double a[], double x, int degree) {
    long i;
    double result = a[0];
    double xpwr = x;
    for (i=1; i<=degree; i++) {
        result += a[i] * xpwr;
        xpwr = x * xpwr;
    }
    return result;
}
```

The `poly` function evaluates a polynomial whose coefficients are given by `a`, i.e.

$$a[0] + a[1]x + a[2]x^2 + a[3]x^3 + \ldots + a[n]x^n$$

where n is the `degree` of the polynomial.

4.2.1 Horner's Method

Horner's Method is an algorithm for polynomial evaluation that reduces the number of operations performed. Given a polynomial

$$p(x) = \sum_{i=0}^{n} a_i x^i$$

Horner's Method factors the polynomial into the new expression

$$p(x) = a_0 + x(a_1 + x(a_2 + \ldots + x(a_{n-1} + a_n x)\ldots))$$

This technique should serve as a starting point for your improved `poly` implementation.

4.3 Testing

There are several files in the *perf* directory, but you only need to worry about filling in *poly.c*. You can look at the implementation of `poly_naive` in *poly_naive.c*. After building your code, you can run

```
./poly <degree> <iters> [-c] [-a]
```

- If the optional `-c` (check) flag is provided, these programs will compare the output of your code against the output of the corresponding naive solution. Consequently, *do not edit the naive versions of these functions*. High performance is meaningless if your program output is not correct, and editing the naive functions could cause your code to appear correct even if it is not.
- If the optional `-a` (all) flag is provided, the speeds of your implementation, the naive implementation, and the fast implementation will be printed.

You will be graded on this project based on the following categories.

- **Functionality**
  - Your code should correctly compute the `poly` function's output.
  - **Note:** you don't have to perform error-checking on the arguments passed to `poly` — this is done for you by the support code.
- **Performance**
  - `poly` runs quickly. The faster your program runs, the more points you will receive.
  - You will not receive any performance points if your code does not produce correct output.
- **Explanation**:
  - You should explain all optimizations that you made in `poly` in your *README*. This will be a significant portion of your grade, so make sure to explain what optimizations you made and why they make your program faster. If you considered any alternatives, explain why you chose to implement your function the way you did and what tradeoffs you may have made.

5 Handing In

To hand in your project, run the command:

```
cs033_handin stringsperf
```

from your *stringsperf* directory.

Your handin must be contain two directories — *strings* and *perf* — that separate your work for the two parts of this project. The directories should be organized as follows:

- *strings* directory
  - All *.c* and *.h* files containing code you have written for the Strings portion of this assignment.
  - A *README* explaining your approaches to `strstr` and `strtok`. You should also document any known bugs and collaborators.
- *perf* directory
  - All *.c* and *.h* files containing code you have written for the Performance portion of this assignment.
  - A *README* explaining your explanation of your `poly` optimizations, as described in the Performance section of the handout. You should also document any known bugs and collaborators.

If you wish to change your handin, you can do so by re-running the handin script. Only the most recent handin will be graded.