Assignment 2: String me along

Due: Mon Oct 16 11:59 pm
Ontime bonus 5%. Grace period for late submissions until Wed Oct 18 11:59 pm

Assignment by Julie Zelenski and Chris Gregg

Learning goals

This assignment is designed to give you

- practice with C-strings (both raw manipulation and using string library functions)
- an opportunity to view Unix utility programs from an internal perspective, as implementer not just client
- exposure to interacting with the Unix filesystem and shell environment variables

Overview

Watch video walkthrough! (https://youtu.be/AAHFpoyk-sM)

This assignment consists of some code-study exercises and a small program to write. Two of the code excerpts come from the C standard library (`atoi` and `strtok`) and the third introduces you to the `opendir` and `readdir` functions.

The program you will write is a version of the Unix `which` command, a utility used to locate and identify executable programs to run. This is an especially apropos way to be introduced to C and Unix; not only does it continue a thread you began in assign0, but implementing the Unix operating system and its command-line tools were what motivated the creation of the C language in the first place! Implementing such a utility program is a very natural use of C, and you'll see how comfortably it fits in this role.

Get started

Check out the starter project from your cs107 repo using the command

```
git clone /afs/ir/class/cs107/repos/assign2/$USER assign2
```

The starter project contains `code.c` with the code for the code-reading exercises, C files `tokenize.c` , `scan_token.c` , and `mywhich.c` , and the supporting `Makefile` , `custom_tests` , and `readme.txt` files. In the `samples` subdirectory, you'll find our sample solutions.

1. Code study: atoi

In assign0, you used the `atoi` function to convert command line arguments from strings to integers. The function name comes from `ascii to integer` , and as you found out in assign0, it is not particularly robust -- it bails at the first non-digit character with no indication the conversion failed. `atoi` has largely been superseded by the more full-featured `strtol` (used in assign1), but we chose the `atoi` implementation as the easier one to study.

Below is an implementation of `atoi` that uses pointer increment to advance through the input string. Although it is probably cleaner to use array indexing, if you read much C code, you will see plenty of pointer arithmetic, so you should get used to reading and understanding it.

```
int atoi(const char *s)
{
    int n=0, neg=0;
    while (isspace(*s)) s++;
    switch (*s) {
        case '-': neg=1;
        case '+': s++;
    }
    /* Compute n as a negative number to avoid overflow on INT_MIN */
    while (isdigit(*s))
        n = 10*n - (*s++ - '0');
    return neg ? n : -n;
}
```

In your `readme.txt` file for assign2, answer the following questions:

a. How is a single digit character converted to its numeric value?

b. If the string begins with a leading minus, at what point does it advance `s` past that char? (Look closely! How the control flows is subtle and easily overlooked)
c. The loop builds up the number as a negative value and later negates it. The comment indicates this choice avoids overflow on INT_MIN. Why does INT_MIN necessitate such a special case?
d. Below are five invalid calls to `atoi`. For each call, work out what is returned and then verify that your understanding is correct by running the program in `code.c`. In your readme.txt, indicate what is returned for each call and explain why.

```
atoi(" $5");
atoi("12:34");
atoi("-2147483649");
int num = 50;
atoi(&num);
atoi(num);
```

2. Code study: strtok

A common string-handling need is to split a string into "tokens" which are separated by one or more delimiting characters. The `strtok` function is the C standard library function used to split strings. However, unlike functions you may have used in CS106B (e.g., `stringSplit`), a single call to `strtok` does not return a nicely formatted vector of the tokens. Rather, you must call `strtok` repeatedly, each time receiving a single token, and continue until `strtok` returns `NULL` to indicate there are no more tokens.

Start by reading the function's man page (`man strtok`) to understand its basic operation. Be sure to read the BUGS section of the man page where it critiques the awkward design of this function.

The first peculiar feature of `strtok` is that it destructively modifies the input string. Rather than construct a new substring for each token, it overwrites the token's ending delimiter in the input string with a null byte, thus re-purposing the existing sequence of characters from the input string to become the token substring without copying those characters to new memory.

Another odd design decision is that `strtok` keeps track of the current state of the tokenization process on behalf of future calls to the function. The first time you call `strtok` on a string to tokenize, you pass the input string as the first argument, but for subsequent calls to `strtok` you pass `NULL` as the first argument. `strtok` tracks the input being tokenized by internally maintaining a pointer to the beginning of the next token. This variable is declared with the `static` qualifier, the purpose of `static` will be explored in a question below.

The following is musl (https://www.musl-libc.org)'s implementation of the `strtok` function:

```
1    char *strtok(char *s, const char *sep)
2    {
3         static char *p = NULL;
4
5         if (s == NULL && ((s = p) == NULL))
6             return NULL;
7         s += strspn(s, sep);
8         if (!*s)
9             return p = NULL;
10        p = s + strcspn(s, sep);
11        if (*p)
12            *p++ = '\0';
13        else
14             p = NULL;
15        return s;
16    }
```

In your `readme.txt` file for assign2, answer the following questions:

a. This is likely the first time you have seen the `static` qualifier applied to a local variable. The Wikipedia article on static variables (https://en.wikipedia.org/wiki/Static_variable) provides an overview of the static qualifier, and the Scope and Example (https://en.wikipedia.org/wiki/Static_variable#Scope) section demonstrates an example of a static local variable and design rationale . Why does `strtok` declare the local variable `p` as `static`?
b. Changing the initialization to `static char *p = s` and re-compiling will produce a compiler error. What is the error message? Use your C reference or web search to

research this error message and how it relates to static variables. In readme.txt, explain how static variables are initialized and how it differs from non-static local variables.

c. The first time `strtok` is called, the input string is passed as the first argument. On subsequent calls to continue tokenizing the same string, `NULL` is passed as the first argument. Given this context, when will the test in line (5) evaluate to true?
d. Read the man page for `strspn` and explain what happens on line (7) when the remaining part of the input string consists of only delimiter characters.
e. Explain what line (12) accomplishes and what this does to the input string.

3. Write scan_token

With the goal of making an improved function that performs the same type of tokenization as `strtok` without its awkward design, you are to write the function `scan_token`. You will write and test this function in isolation now, and then will use the function later when writing the `mywhich` program. The required prototype for `scan_token` is:

```
bool scan_token(const char **p_input, const char *delimiters,
                char buf[], size_t buflen);
```

The function scans the input string to determine the extent of the token using the delimiters as separators and then writes the token characters to `buf`, making sure to terminate with a null char. The function returns `true` if a token was written to `buf`, and `false` otherwise.

Specific details of the function's operation:

- Your implementation of `scan_token` should take the same general approach as `strtok`, meaning it can (and should!) use the handy `<string.h>` functions such as `strspn` and `strcspn`, but it should not replicate the bad parts of its design, which is to say no static variables, no weird use of the input argument to pass information across a sequence of calls, and should not destroy the input string.
- The function separates the input into tokens in the same way that `strtok` does: it scans the input string to find the first character **not** contained in `delimiters`. This is the beginning of the token. It scans from there to find the first character contained in `delimiters`. This delimiter (or the end of the string if no delimiter was found) marks the end of the token.
- Note that the parameter `p_input` is a `char **`. This is a pointer argument that is being passed by reference. The client passes a pointer to the pointer to the first char of the input string. The function will update the pointer held by `p_input` to point to the next character following the token that was just scanned.
- `buf` is a fixed-length array to store the token and `buflen` is the length of the buffer. `scan_token` should not write past the end of `buf`. If a token does not fit in `buf`, the function should write `buflen - 1` characters into `buf`, write a null byte in the last slot, and the pointer held by `p_input` should be updated to point to the next character following the `buflen - 1` characters in the token. In other words, the next token scanned will start at the first character that would have overflowed `buf`.

Consider this sample use of `scan_token`:

```
const char *input = "super-duper-awesome-magnificent";
char buf[10];
const char *remaining = input;

while (scan_token(&remaining, "-", buf, sizeof(buf))) {
    printf("Next token: %s\n", buf);
}
```

Running the above code produces this output:

```
Next token: super
Next token: duper
Next token: awesome
Next token: magnifice
Next token: nt
```

Write your implementation of `scan_token` in the `scan_token.c` file. You can test it using our provided `tokenize.c` program. The `tokenize` program is integrated with sanitycheck.

In addition to teaching you the inner workings of computer systems, CS107 also provides strength+cardio training for your coding skills. A key piece of this training is learning the value of thoughtful and thorough testing--better for you to find the bugs than our autograder! The default

tests supplied with sanitycheck are a start but these basic tests are not comprehensive and should be supplemented with your own tests. In order to encourage you to do the careful testing that we hope you would do anyway, for assign2 we require that you submit your sanitycheck `custom_tests` file with at least **5 thoughtful and varied tests of your own for the tokenize program**. These tests should cover a variety of cases that validate that `scan_token` is working properly on ordinary cases as well on on inputs that are unusual or edge conditions.

4. Code study: opendir/readdir

Many Unix utilities read and write from the filesystem. The `<dirent.h>` header file provides functions used to access to the contents of directories. In particular, you will be using the `opendir` and `readdir` functions in your `mywhich` program to get information about the files in a given directory.

The `<dirent.h>` header defines two important data types:

- `DIR` : a type representing a directory stream
- `struct dirent` : a record of information for one file or directory (name, number, type, and so on). Read the man page for readdir (`man readdir`) to see the struct definition and its documentation.

The easiest way to see how to gather directory information is through an example:

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

void list_filenames(const char *dirpath)
{
    DIR *dp = opendir(dirpath);
    if (dp == NULL) return;

                    // loop through directory entries
    struct dirent *entry;
    while ((entry = readdir(dp)) != NULL) {
        printf("%s\n",entry->d_name);
    }

    closedir(dp);
}
```

Notes:

- If you are used to C++, the `struct dirent *entry;` line might look a bit funny. In C, unless `struct` s are `typedef` 'd, you need to declare a variable using the `struct` tag. As with C++, structs access their members with dot or arrow notation, as in `entry->d_name` in the program above.
- You must call the `closedir` function after you are done with a `DIR` pointer to release its resources. The `opendir` call both allocates dynamic memory and uses an entry in the file table. If you forget to close the `DIR` , those resources cannot be reclaimed. Run the `code.c` program under `valgrind` with and without the call to `closedir(dp)` and see what Valgrind has to say about this.

In your readme.txt file, answer the following questions:

a. What does `struct dirent` define to be the maximum filename length?
b. How many bytes of memory does Valgrind report are "lost" if a program does an `opendir` without a matching `closedir` ?

Review and comment starter code

The file `mywhich.c` is given to you with an incomplete main function that sketches the expected behavior for the case when mywhich is invoked with no arguments. You are to first read and understand this code, work out how to change/extend it to suit your needs, and finally add comments to document your strategy.

Some questions you might consider for self-test: (do not submit answers)

- What is the third argument to main? How do you determine the end of the `envp` array?
- What is `PATH_MAX` ? What is it used for?
- If the user's environment does not contain a value for `MYPATH` , what does `mywhich` use instead?

- Do you see anything unexpected or erroneous? *We intend for our code to be bug-free; if you find otherwise, please let us know!*

As per usual, the code we provide has been stripped of its comments and it will be your job to provide the missing documentation.

5. Implement the `mywhich` program

What does the `which` command do?

The `which` command searches for a command by name and reports where its matching executable file was found. Read its man page (`man which`) and try it out, e.g. `which ls` or `which make` or `which vim` . The response from `which` is the full path to the matching executable file or no output if not found.

It may not be obvious at first, but this search is intimately related to how commands are executed by the shell. When you run a command such as `ls` or `vim` , the shell searches for an executable program that matches that command name and then runs that program.

Where does it search for executables? You might imagine that it looks for an executable file named `vim` in every directory on the entire filesystem, but such an exhaustive search would be both incredibly inefficient and dangerously insecure. Instead, it searches only those directories that have been explicitly listed in the user's search path. The default search path includes directories such as `/usr/local/bin/` and `/usr/bin/` which house the executable files for the standard unix commands. (The name `bin` is a nod to the fact that executable files are encoded in **bin**ary).

The user can configure their search path by changing the value of their `PATH` environment variable. As you saw in lab2, environment variables track information like username (`USER=zelenski`) and the user's shell (`SHELL=/bin/bash`). The environment variable of particular interest for `which` is `PATH=/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/sbin:/sbin:/usr/games` . The value for `PATH` is a sequence of directories separated by colons; these are the directories searched when looking for an executable. When looking for a command, `which` searches the directories in the order they are listed in the search path and stops at the first directory that contains a matching executable. In order to match, the file's name must be an exact match and the file must be readable and executable by the user.

How does `mywhich` operate?

The `mywhich` program you are to write is similar in operation to the standard `which` with these differences:

- `mywhich` uses the environment variable `MYPATH` for the search path. If no such environment variable exists, it falls back to `PATH` . (Standard `which` always uses `PATH` as the search path)
- `mywhich` invoked with no arguments prints the list of directories searched. (standard `which` with no arguments does nothing)
- `mywhich` treats each command-line argument prefixed with a `+` as a wildcard match. (standard `which` has no option for wildcard match)
- `mywhich` does not support any command-line flags. (standard `which –a` prints all exact matches)

When invoked with **no arguments**, `mywhich` prints the directories in the search path, one directory per line. This use case is a testing aid to verify that you are accessing the correct environment variable and can properly tokenize it.

```
myth> ./mywhich
Directories in search path:
/usr/bin
/usr/local/bin
/usr/pubsw/bin
/bin
```

When invoked with **one or more arguments**, `mywhich` searches the directories in the search path for an exact match for each argument. The sample output below shows invoking `mywhich` to find three executables. Two of them were found, but no executable named `submit` was found in any directory in the user's MYPATH and thus nothing was printed for it.

```
myth> ./mywhich xemacs submit cp
/usr/bin/xemacs
/bin/cp
```

Any **argument prefixed with +** is handled as a wildcard search, instead of an exact match. A wildcard search prints all executables that contain that pattern from all directories in the search path. Let's say you vaguely remember there is a "fun" unix command, so use a wildcard search to find it:

```
myth> ./mywhich +fun
/usr/bin/funzip
/usr/bin/pdfunite
```

For testing purposes, you should test having run mywhich with different directories in the search path. Rather than muck with your actual `PATH` (which can create total chaos), we recommend that you change `MYPATH`, which only affects `mywhich` and nothing else. Use `env` to set the value of `MYPATH` when running `mywhich` like this:

```
    myth> env MYPATH=/tmp:tools ./mywhich submit
    tools/submit
```

Requirements for `mywhich`

- **Usage**. The `mywhich` program is invoked with zero or more arguments. Any argument prefixed with + is handled as a wildcard search. All other arguments are searched for using exact match. An argument is a non-empty string of one or more characters, i..e. `""` or just a single `+` is invalid.
- **Assumptions.** You may assume correct usage in all cases and that the user's `MYPATH` and `PATH` variables are well-formed sequences of one or more paths separated by colons. You do not need to detect or cope with situations where these assumptions do not hold and we will not test on any inputs that violate these assumptions, e.g. no usage of unsupported `–a` flag, no empty arguments, and no malformed values for `MYPATH`.
- **Operation**. The user's `MYPATH` (or `PATH` if there is no `MYPATH` variable in the user's environment) defines the search path. The directories are searched in the order they are listed in the search path. For an exact match, the search stops at the first directory containing a readable, executable file matching the command name. For a wildcard search, it searches all directories and prints all matching executables.
- **Expected output.** For each command name, it prints the full path to the first matching executable or nothing if no matching executable was found. The matched executables are listed in the order that the command names were specified on the command-line. For a wildcard search, it prints the full path for every matching executable in any of the directories in search path. The directories are searched in order of the search path, but the matching files from the directory may be printed in any order.
- **Restrictions**. Your own code should manually search the environment using the `envp` argument to `main`. Your code is **prohibited from using facilities such as** `getenv`, `env`, and `which` to do this work on your behalf.

Advice/FAQ

Don't miss out on the good stuff in our companion document!

Go to advice/FAQ page (advice.html)

Grading

Have you read how assignments are graded (/class/cs107/advice/assigngrade.html)? For this assignment, the anticipated point breakdown will be in the neighborhood of:

Readme questions (35 points)

- **readme.txt**. (30 points) For the code reading questions, you will be graded on the understanding of the issues demonstrated by your answers and the correctness of your conclusions.
- **custom_tests**. (5 points) These points reward your effort in identifying cases that provide comprehensive test coverage for the tokenize program. We are looking for at least 5 thoughtful tests that cover a variety of inputs, including edge conditions.

Functionality (65 points)

- **Sanity cases** (25 points) Correct results on the default sanity check tests.
- **Comprehensive/stress cases** (30 points) Correct results for additional test cases with broad, comprehensive coverage and larger, more complex inputs.
- **Clean compile** (2 points) Compiles cleanly with no warnings.
- **Clean run under valgrind** (6 points) Clean memory report(s) when run under valgrind. Memory errors (invalid read/write, use of freed memory, etc) are significant deductions. Memory leaks are a minor deduction. Every normal execution path is expected to run cleanly with no memory errors nor leaks reported. We will not test exceptional/error cases under Valgrind.

Code review (buckets together weighted to contribute ~15 points)

- *Use of pointers and memory*. We expect you to show proficiency in handling pointers/memory, no unnecessary levels of indirection, correct use of pointee types and typecasts, and so on. For this program, you should not need and should not use dynamic memory (i.e. no malloc/strdup).
- *Program design*. We expect your code to show thoughtful design and appropriate decomposition. Data should be logically structured and accessed. Control flow should be clear and direct. When you need the same code in more than one place, you should unify, not copy and paste. If the C library provides functionality needed for a task, you should leverage these library functions rather than re-implement that functionality.
- *Style and readability*. We expect your code to be clean and readable. We will look for descriptive names, defined constants (not magic numbers!), and consistent layout. Be sure to use the most clear and direct C syntax and constructs available to you.
- *Documentation*. You are to document both the code you wrote and what we provided. We expect program overview and per-function comments that explain the overall design along with sparing use of inline comments to draw attention to noteworthy details or shed light on a dense or obscure passage. The audience for the comments is your C-savvy peer.

On-time bonus (+5%)

The on-time bonus for this assignment is 5%. Submissions received by the due date earn the on-time bonus. The bonus is calculated as a percentage of the point score earned by the submission.

Finish and submit

Review the How to Submit (/class/cs107/submit.html) page for instructions. Submissions received by the due date receive the on-time bonus. If you miss the due date, late work may be submitted during the grace period without penalty. No submissions will be accepted after the grace period ends, please plan accordingly!

How did it go for you? Review the post-task self-check (/class/cs107/selfcheck.html#assign2).