CSC 352, Fall 2015
Assignment 9
Due: Friday, November 6 at 23:59:59

The Usual Stuff

All the usual stuff on assignments applies to this one, too: make an `a`*`N`* symlink, use our `gcc` alias, use the Tester, the Tester runs `gcc` first, etc. Refer to previous write-ups if you need a refresher.

This is an in-between assignment

We've covered arrays of pointers, and pointers to pointers, but not memory allocation, so to work with what we know, this is a collection of simple problems that work with the command-line arguments, which are provided to `main` as a zero-terminated array of pointers. See C slides 332-333 and, for more, 5.1.2.2.1 in the C11 standard. One thing the slides don't mention is that the array of argument pointers and the strings they point to can be modified.

Point values are a little inflated for these four problems; don't leave any of these easy points on the table!

No restrictions, but...

In short, there are no restrictions on your solutions for this assignment.

We'll soon be studying dynamic memory allocation but because these problems were written to not need dynamic memory allocation, I encourage you to think in terms of solutions that don't require allocation.

The only library routines my current solutions use are `atoi`, `exit`, `gets`, `printf`, `putchar`, `puts`, `strncmp`, and `strstr`, but you're free to use any C library routines that you wish.

More `<string.h>` routines

I mention above that I use `strncmp` and `strstr`, two routines you haven't seen yet. Here's the prototype for `strncmp` that `man strncmp` shows:

```
int strncmp(const char *s1, const char *s2, size_t n);
```

`strncmp` compares the first n characters of `s1` and `s2`; you can read the man page for the details but what I want to point out is that `const` keyword. It's essentially a promise that `strncmp` won't change the characters that are pointed to by `s1` and `s2`. We'll learn all about `const` soon.

You might expect `strncmp` to return 1 when the strings are equal but, like Java's `String.compareTo`, 0 is returned when the strings are equal. See the man page for details.

Problem 1. (6 points) `ilv.c`

For this problem you are to write a C program, `ilv.c`, that reads lines from standard input and interleaves zero or more copies of each input line with the command line arguments. Here's an example:

```
% cat a9/ilv.1
Here
is some
text!
% ilv X - Y < a9/ilv.1
XHereY
Xis someY
Xtext!Y
%
```

In this case, `ilv` has three command line arguments: `"X"`, `"-"`, and `"Y"`. The dash (minus sign, '-') stands for the text of the input line. Each line of the input is read and printed, preceded by an `"X"` and followed by a `"Y"`.

In the following example a sequence of `"mv"` commands is generated from a list of file names:

```
% cat a9/ilv.2
x.c
y.java
a1.notes
c.notes
% ilv  "mv "  -  " "  -  ".old"  < a9/ilv.2
mv x.c x.c.old
mv y.java y.java.old
mv a1.notes a1.notes.old
mv c.notes c.notes.old
%
```

Note that two dashes are used to produce two separate copies of the input line, and that quoted arguments are used to inject blanks into the output.

`ilv` treats command line arguments very simply: A one-character argument that is a minus sign stands for a copy of the input line. Everything else is treated as plain text, with one exception. Here's the exception: the argument `"--"` (two dashes) indicates that a single dash should be output.

Example:

```
% echo Testing | ilv --  -  -  --  -
-TestingTesting-Testing
% echo Testing | ilv x- - -x
x-Testing-x
%
```

Behavior is undefined if `ilv` is given no arguments.

Implementation note: Assume that lines are less than 1000 characters in length. That implies that given `char line[1000]`, you can read a line with `gets(line)`.

Problem 2. (7 points) `gdiff.c`

`gdiff` takes two groups of command line arguments, separated by a dash (minus sign, '−') and outputs the elements in the first group that do not appear in the second group. Here's an example of operation:

```
% gdiff a b c d e - e a x
b
c
d
```

The first group is `{a,b,c,d,e}`. The second group, following the dash, is `{e,a,x}`. The output is `b`, `c`, and `d`, one per line, because those arguments do not appear in the second group.

There may be arguments that appear only in the second group, such as `x` above. They have no effect.

More examples:

```
% gdiff red blue green blue red - red green
blue
blue
% gdiff a aa b c a b - a b
aa
c
% gdiff $(seq 10) - 9 3 6
1
2
4
5
7
8
10
%
```

If you've had CSC 245, you'll note that this operation is similar to set difference but because there may be duplicates, the groups aren't sets.

Either or both groups might be empty but it's an error if no dash is specified. Behavior is undefined if more than one argument is a dash.

```
% gdiff testing - this
testing
% gdiff testing -
testing
% gdiff - testing
% gdiff testing
Error: no '-' specified
%
```

Note that `gdiff` can be combined with wildcards and command substitution to do useful things. For example, `wc $(gdiff * - *.o *[0-9])` runs `wc` on everything except `.o` files and files that end with a digit.

Problem 3. (9 points) `ag.c`

`ag.c` is a recycle/extension of `argGrep.java` from the mid-term exam. `ag` looks for occurrences of the first argument in subsequent arguments. It prints matching arguments on a single line, and below that line, prints carets to mark the first occurrence in each argument.

```
% ag in testing looks inward
testing inward
    ^^  ^^
% ag test testing the tests of the tester
testing tests tester
^^^^    ^^^^  ^^^^
% ag t testing the tests of the tester
testing the tests the tester
^       ^   ^     ^   ^
% ag x testing the tests of the tester
%
```

Behavior is undefined if `ag` is run with less than two arguments.

Implementation note: My solution uses the `strstr` library routine. (Do `man strstr`.)

Problem 4. (14 points) `evalit.c`

`evalit.c` is similar to `eval.java` from assignment 2: it evaluates expressions specified as command line arguments but the details are a little different. I would have called it `eval.c` but there's already `eval(1)`, and I didn't want to collide with it.

`evalit` is invoked with a command line that specifies zero or more variable initializations followed by an expression to evaluate. Here are some examples:

```
% evalit width=5 height=7 width*height+10
width*height+10 is 45 (given width=5, height=7)
% evalit 20+30*40/50-60
20+30*40/50-60 is -20
% evalit i=1 v=5 x=10 c=100 i+i+i*v*x*c
i+i+i*v*x*c is 15000 (given i=1, v=5, x=10, c=100)
%
```

Note that each initialization and the expression itself is an argument. If any variables are specified, a "`(given ...)`" clause is added.

Like `eval.java`, there is no precedence—expressions are simply evaluated left to right. The binary operators `+`, `*`, `-`, and `/` are supported. Only integers are supported, and all computations are done with integers. Unary negation is <u>not</u> supported. Variables consist of only lower-case lettters.

An initialization must be present for every variable. If not, an error is printed:

```
% evalit x
variable 'x' is undefined
%
```

`evalit` can handle an unlimited number of variable initializations and an arbitrarily long expression.

Aside from detecting undefined variables, `evalit` does no error checking whatsoever. Assume everything is well-formed.

Implementation notes

Use `int atoi(const char *s)` to convert a character string to an `int`. There's a man page for `atoi` (and more), but you can see an example of an `atoi` call way back on slide 15.

It's good to understand that with the ASCII character set you can see if a character is a letter by seeing if its value is between `'a'` and `'z'`, and so forth, but a better, and portable, way to do that sort of thing is to use the "ctype" macros. If you do `man islower`, you'll see `islower(c)`, `isdigit(c)`, and more.

You might be tempted to build some data structures, maybe use `malloc`, and/or even look ahead to C `structs` to handle the variables, but I don't do any of that. I use the argument list pretty much as is to look up values for variables. I recommend that you strive for a similarly simple solution.

I'll put some suggestions about helper functions in `a9/evalit-hints`. Remind me by mail if that file hasn't appeared by noon on Sunday, November 1.

A pitfall with wildcards

You have noticed that some examples above use an asterisk, a bash wildcard, without any quotes. Those examples rely on the fact that there are no files in the current directory that match those patterns, and the string gets passed unchanged as an argument. However, an expression like `x*x` would match the name `x.pptx`. With `x.pptx` in the current directory, I see this:

```
% evalit x=3 x*x
variable 'pptx' is undefined
```

`echo` shows what `evalit` is getting:

```
% echo evalit x=3 x*x
evalit x=3 x.pptx
```

Adding quotes makes things, work, of course:

```
% evalit x=3 "x*x"
x*x is 9 (given x=3)
```

Problem 5. Extra Credit `observations.txt`

Submit a plain text file named `observations.txt` with...

(a) (1 point extra credit) An estimate of how long it took you to complete this assignment. To facilitate programmatic extraction of the hours from all submissions have an estimate of hours on a line by itself, more or less like one of the following three examples:

```
Hours: 6
Hours: 3-4.5
Hours: ~8
```

If you want the one-point bonus, be sure to report your total (estimated) hours on a line that starts with `"Hours:"`. There must be only one `"Hours:"` line in `observations.txt`. It's fine if you care to provide per-problem times, and that data is useful to us, but report it in some form of your own invention, not with multiple lines that contain "Hours:", in either upper- or lower-case.

Other comments about the assignment are welcome, too. Was it too long, too hard, too detailed? Speak up! I appreciate all feedback, favorable or not.

(b) (1-3 points extra credit) Cite an interesting course-related observation (or observations) that you made while working on the assignment. The observation should have at least a little bit of depth. Think of me saying "Good!" as one point, "Excellent!" as two points, and "Wow!" as three points. I'm looking for quality, not quantity.

Turning in your work

Use `a9/turnin` to submit your work. Each run creates a time-stamped "tar file" in your current directory with a name like `a`*`N`*`.`*`YYYYMMDD`*`.`*`HHMMSS`*`.tz` You can run `a9/turnin` as often as you want. We'll grade your final submission.

Note that each of the `aN.*.tz` files is essentially a backup, too, but perhaps mail to `352f15` if you need to recover a file—it's easy to accidentally overwrite your latest copies with a poorly specified extraction.

`a9/turnin -l` shows your submissions.

To give you an idea about the size of my solutions, here's what I see as of press time:

```
% wc $(grep -v txt < a9/delivs)
  21   44  421 ilv.c
  36   84  692 gdiff.c
  38  124  919 ag.c
  92  209 1835 evalit.c
 187  461 3867 total

% for i in $(grep -v txt a9/delivs); do echo $i: $(tr -dc \; < $i | wc
-c); done
ilv.c: 7
gdiff.c: 14
```

```
ag.c: 26
evalit.c: 41
```

There's only a comment or two in my code.

Miscellaneous

This assignment is based on the material on C slides 1-339.

Point values of problems correspond directly to assignment points in the syllabus. For example, a 10-point problem corresponds to 1% of your final grade in the course.

Feel free to use comments to document your code as you see fit, but note that no comments are required, and no points will be awarded for documentation itself. (In other words, no part of your score will be based on documentation.)

Remember that late assignments are not accepted and that there are no late days; but if circumstances beyond your control interfere with your work on this assignment, there may be grounds for an extension. See the syllabus for details.

My estimate is that a student who has only taken CSC 127A and 127B but done well in them and has completed the previous assignments will need 5-7 hours to complete this assignment.

Keep in mind the point value of each problem; don't invest an inordinate amount of time in a problem or become incredibly frustrated before you ask for a hint or help. Remember that the purpose of the assignments is to build understanding of the course material by applying it to solve problems.

If you put five hours into this assignment and don't seem to be close to completing it, it's probably time to touch base with us. Specifically mention that you've reached five hours. Give us a chance to speed you up!

Our goal is that everybody gets 100% on this assignment AND gets it done in an amount of time that is reasonable for them.

I hate to have to mention it but keep in mind that cheaters don't get a second chance. If you give your code to somebody else and they turn it in, you'll both likely fail the class, and more. (See the syllabus for the details.)