PS01 - `thecho` - "Tar Heel `echo`"

Little Languages - COMP590 - Spring 2019

Overview

In this problem set you will implement a simplified version of the `echo` utility program. Yours will be named `thecho`, short for "Tar Heel Echo".

The purpose of this problem set is to better understand command-line arguments, common conventions around program *flags* or *options*, and begin thinking about string interpretation.

The `echo` utility is included standard on every *nix machine. Its purpose is to print out the command-line arguments it receives when the program is run. Soon you will learn some good uses for `echo` in conjunction with environment variables, shell piping, and output redirection. For now, it is simply printing out its inputs.

Grading

Autograding Levels

0. 40 points - Baseline Functionality - No option flags.
1. 20 points - Supporting the `-n` flag.
2. 15 points - Supporting the `-e` flag with `\t` `\n` and `\\` escape sequences.
3. 5 points - Ensuring both flags can be composed together.

Hand-graded Points

1. 10 points - Style and Documentation

- Did you run `:RustFmt` to ensure proper indentation and formatting?
- Did you name variables meaningfully?
- Did you add comments to segments of code that are not self-documenting?

2. 10 points - Level 2 Iteration

- Did you iterate over characters to process the escape sequences for the `-e` flag? More specifically, did you *not* use any built-in methods like `replace`.
- Was your implementation efficient and avoided unnecessary, redundant work?

No external Rust crates are allowed for this assignment. There are great crates for command-line argument parsing and we will begin making use of them soon. It is important for you to understand the fundamentals of command line argument processing before relying upon 3rd party libraries.

Example Use Cases

Before you begin implementing `thecho`, you should get a feel for the standard `echo` program's equivalent functionality. In your virtual machine, try experimenting in the shell. You'll type the lines following the $ prompt. Remember, your program will be called `thecho`.

Level 0 Functionality

The simplest use case for `echo` is it prints inputs exactly as given.

```
$ echo hello
hello
$ echo hello world
hello world
$ echo "hello\nworld"
hello\nworld
$
```

Level 1 - Support for the `-n` Flag

Open the manual for `echo`, using the shell command `man echo`, and read the description of its short flag `-n`. Try experimenting with the flag as shown below. These arguments are called *flags* or *options* because they change the default behavior of a program. Note the flag must come before any other text.

```
$ echo hello -n world
hello -n world
$ echo -n hello world
hello world$
```

Level 2 - Support for the `-e` Flag

Open the manual for `echo`, using the shell command `man echo`, and read the description for the `-e` flag as well as the sequences that are in effect when you use the `-e` flag. Your program only needs to handle the newline, tab, and backslash escapes. For all others it should panic.

Notice for the examples where escape codes are used below that the arguments are surrounded by quotes. Doing so tells the shell to ignore the escape codes and send them to your program. If you leave off the quotes, then the shell will interpret the escape codes differently before calling your program.

```
$ echo "hello\tworld\n...\\"
hello\tworld\n...\\
$ echo -e "hello\tworld\n...\\"
hello   world
...\
$ echo "hello\tworld" -e
hello\tworld -e
```

Level 3 - Ensuring `-n` and `-e` Compose Together

You should be able to specify the flags in either order. They must occur as the first arguments to the program. Otherwise, if they follow non-flags, they will be ignored, as shown in the first example below.

```
$ echo hello -n -e world
hello -n -e world
$ echo -n -e "hello\nworld\t"
hello
world   $ echo -e -n "hello\nworld\t"
hello
world   $
```

Differences from the standard `echo`

When your program encounters an unspecified escape sequence using the `-e` flag, it should `panic!` with any error message. The standard `echo` has more flags and options than yours will, as you saw in the manual pages for it.

Getting Started

GitHub Classroom Starter URL: https://classroom.github.com/a/F8Ck1a7f

Cloning the Repository

Please follow the link above to setup your repository for this problem set. Once you've done so, you'll want to find its Clone link and be sure you choose the SSH option that begins with `git@github.com:comp590-...`. Copy that link to your clipboard.

While logged in to your VM in a terminal, issue the following commands and replace the words in <tags> with the values specific to you:

```
$ cd ~
$ git clone <paste>
$ cd ps01-thecho-<your-github-username>
```

You can now begin working on `thecho` by editing `src/main.rs` in `vim`:

```
$ vim src/main.rs
```

Running `thecho` in Development

It's handy to be able to run your program in `vim` without having to quit to the shell every time you make a change.

To run a shell command while working in `vim`, enter Ex/Ed mode from Normal mode by typing a colon character. Then, add an exclamation point. Finally, type your shell program. The output of your command will be displayed and when you press enter you will be taken back to `vim`.

For example, from normal mode try typing: `:!echo hello world`.

There are two ways of building and running `thecho`.

Using `cargo build` and running `thecho`

The `cargo build` command will build your project and results in the compiled `thecho` program in the folder `target/debug` directory. This directory is a part of your shell's program lookup path meaning that if you now run the command `thecho` it'll find your compiled program while you're working in this directory. For example, in `vim`, you would run:

```
:!cargo build
:!thecho hello world
```

This sequence of steps rebuilds your project and runs it with the command line arguments `hello world`.

Using `cargo run`

The `cargo run` command combines the two steps of the previous example into a single step. It has one caveat that you need to understand to make sense of it, though. When passing command-line arguments to your program, you need to add two dashes `--` after `cargo run` and before your arguments. For example:

```
:!cargo run -- hello world
:!cargo run -- -n hello world
:!cargo run -- -n -e "hello\\world"
```

The reason for this is `cargo run` has some of its own flags that can modify its behavior. It uses the `--` to separate the flags you are sending to it versus those you are sending to the `thecho` program. Notice you do not see `thecho` in these commands but behind the scenes it is doing both steps shown in the previous example.

Helpful Hint: The Ed/Ex mode in `vim` keeps a history of the commands you run. If you begin with a `:` colon to enter the mode and then press the up key on your keyboard you will move through your history of recent commands. When you're testing a specific case this is handy for rerunning your last command.

No unit nor functional tests are included with this problem set. You should think through the test cases for each level of the problem set and use either of the methods shown above to convince yourself it works. Later we will discuss how to automate the testing of projects like these.

Making & Pushing Commits

Once your tests are passing, you will want to commit your changes to your project's git repository. We will discuss `git` in much more depth, but for now you can think of it as a backup checkpointing program. The following commands will make a new checkpoint, called a *commit*, in your repository and upload them ("push" them) to your private repository backup on GitHub.

```
git add src/main.rs
git commit -m "Describe your recent changes to the project here."
git push origin master
```

Once you have issued these commands, open your browser to your project on GitHub and notice that the changes you made to the file `src/main.rs` are updated in your repository. You should also see the commit message you added above in the commits history.

6. Submitting to Gradescope

Once your work is pushed to GitHub you can submit to the Gradescope autograder. To do so, open Gradescope and find the project. You can create a submission by selecting your private repository from the Repository selector and `master` from the branch selector. Once uploaded, the autograder should run with feedback shortly.

Using Gradescope, after the late deadline concludes, the COMP590 team will manually grade your code for style and required elements. Please see the rubric on the first page for a sense of what we're looking for.