Build Your Own Shell

This challenge is to build your own shell.

A shell is a program that exposes the operating system's services to us, the user of the system. We're going to focus on the command line versions, like sh, bash, zsh and others.

The Challenge - Building Your Own Shell

I use zsh on my MacBook, here's what the man page says about it:

```
Zsh is a UNIX command interpreter (shell) usable as an
interactive login shell and as a shell script command
processor.
```

We're going to build our shell, which will enable us to run commands on our operating system

Step Zero

Please set up your IDE / editor and programming language of choice and proceed directly to step 1 once you're ready. I've called my project **ccsh** for **C**oding **C**hallenges **Sh**ell.

Step 1

In this step your goal is to create the simples possible command line shell. That's a program that starts up waits for the user to type in a command. To make it easier to see which shell is running I've given my shell a unique prompt: `ccsh>`.

When the user enters a command we'll need to trim any trailing whitespace or newline characters and then spawn a new process to run the entered command.

For example here's what happens when I run my `ccsh`:

```
% ccsh
ccsh> ls
Cargo.lock      Cargo.toml      src             target
%
```

My shell outputs the prompt `ccsh>`. I have then run the command `ls` and it has output the results below the prompt. In this case you see the files from my Rust implementation of **ccsh**.

Note that `ccsh` immediately terminates after running the command, returning me to the normal shell.

Step 2

In this step your goal is to handle multiple commands, to do that you'll want wrap your existing command handling code in a loop. We want that loop to continue indefinitely, but we will also want to be able to exit from the shell.

To allow us to exit the shell most commands offer a builtin command - I bet you can't guess what the command is! 😀

So to complete this step, extend your shell to accept multiple commands until the user enters the builtin command: `exit`.

That should look something like this:

```
% ccsh
ccsh> ls
Cargo.lock      Cargo.toml      src             target
ccsh> pwd
/Users/john/dev/challenge-shell/ccsh
ccsh> exit
%
```

Don't forget to make your shell wait for the running command to finish before trying to capture the next command from a user.

Step 3

In this step your goal is to handle non-existent commands.

Now that we can enter multiple commands and safely exit our shell we want to ensure it is robust. That is if the user tries to run a command that doesn't exist our shell shouldn't crash.

That should look something like this when it's done:

```
% ccsh
ccsh> fubar
No such file or directory (os error 2)
ccsh> exit
%
```

Step 4

In this step your goal is to be able to run external commands that take arguments, for example `ls -la`, or `cat <filename>`.

To do that we'll need to capture the user input and then split it into parts, the command and the arguments to be passed to the command that we spawn. For example I'll run my shell and pass a filename to the `cat` program, which prints out to the terminal the contents of the file:

```
% ccsh
ccsh> cat cargo.lock
# This file is automatically @generated by Cargo.
# It is not intended for manual editing.
version = 3

[[package]]
name = "ccsh"
version = "0.1.0"
ccsh> exit
%
```

Step 5

In this step your goal is to implement a builtin command (`cd`) to change directories and `pwd`, to get the current working directory. The `cd` command has to be built in to the shell because it changes the internal state of the shell.

There are several other builtin commands you might have used regularly: `cd`, `exit`, `export`, `pwd` and `unset`. You can find the full list using the man pages for your shell.

Once you've implemented cd and pwd you should be able to use them as so:

```
% ccsh
ccsh> ls
Cargo.lock      Cargo.toml      src             target
ccsh> cd target
ccsh> ls
CACHEDIR.TAG    debug
ccsh> pwd
/Users/john/dev/challenge-shell/ccsh/target
ccsh> cd ..
ccsh> pwd
/Users/john/dev/challenge-shell/ccsh
ccsh> exit
%
```

Step 6

In this step your goal is to support pipes. If you're read any of the Coding Challenges that build Unix command like tools, you've had seen how much I rely on pipe to chain together simple tools to build complex data munging solutions.

If you're not familiar with the terminology, most shells allow us to 'pipe' the output from one command into the next command using the `|` character.

For example in the build your own cut challenge we piped the output of a `cut` command into `uniq` and then used `wc`:

```
cut -f2 -d, fourchords.csv | uniq | wc -l
```

To do this you'll need to find out how to redirect the standard input and output streams with your chosen programming language.

To test that we've implemented pipes correctly we can do something similar:

```
% ccsh
ccsh> curl https://www.gutenberg.org/cache/epub/132/pg132.txt -o
test.txt
```

```
% Total    % Received % Xferd  Average Speed   Time    Time     Time
Current
Dload  Upload   Total   Spent    Left  Speed
100  333k  100  333k    0     0   366k      0 --:--:-- --:--:-- --:--:-
-  368k
ccsh> cat test.txt | wc -l
7137
ccsh> exit
%
```

Here we downloaded a book from Project Gutenberg to use as a test, saving it as `test.txt`. Then we piped the text of `test.txt` into the command line tool `wc` and counted the number of lines.

Step 7

In this step your goal is to handle signals that would interrupt the shell. For example CTRL-C. When a user enters CTRL-C to quit a program we don't want it to terminate the shell as well.

That said we do still want to be able to quit command's we've run in our shell with CTRL-C, so you will need to ensure the signal handler is restored for child commands.

Here's what that looks like:

```
% ccsh
ccsh> ^C
ccsh> cat
^Cccsh> exit
ccsh %
```

In this example we can see that CTRL-C (rendered as `^C`) does not terminate the shell, but does terminate the child command `cat`.

Step 8

In this step your goal is to add a command history to your shell. You want to have the following features:

1. Save all executed commands to the history.

2. Save the history to disk when the shell exits (to a file, `.ccsh_history` in the users HOME directory.
3. Reload the history from disk when the shell is started.
4. Allow the user to scroll through the history with the up and down arrow keys.
5. Support the built in command: `history` to list the command history.

That might look something like this:

```
ccsh> ls
Cargo.lock      Cargo.toml      src             target
test.txt
ccsh> pwd
/Users/johncrickett/dev/CodingChallengesFYI/challenge-shell/ccsh
ccsh> ls
Cargo.lock      Cargo.toml      src             target
test.txt
ccsh> history
ls
pwd
ls
ccsh> exit
ccsh %
```

Once you've got all that working - congratulations, you've built your own command line shell!

Going Further

If you want to take this further, I'd suggest adding the ability to run shell scripts.

Help Others by Sharing Your Solutions!

If you think your solution is an example other developers can learn from please share it, put it on GitHub, GitLab or elsewhere. Then let me know - ping me a message on the Discord Server or in the Coding Challenges Sub Reddit, via Twitter or LinkedIn or just post about it there and tag me.

If you would like to recieve the coding challenges by email, you can subscribe to the weekly newsletter on SubStack here: