The UNIX Shell

Tristan Miller

February 2026

What is a shell?

- The user interface for an operating system
  - Start, manage, and stop programs
  - Create and manipulate files and directories
  - Monitor the system
- *Graphical shell* vs *command-line shell*

History

- **sh** – created by Ken Thompson at Bell Labs in 1971
- Replaced by the Bourne Shell (**sh**) for Unix 7 in 1979
- This spawned a lot of derivatives
  - **ash** – Almquist shell (BSD, busybox, etc.)
  - **bash** – Bourne Again shell (ubiquitous)
  - **ksh** – Korn shell
  - **zsh** – Z shell (default on macOS and some Linuxes)
- Standardized by POSIX

Variables

- *Stringly*-typed language
- **`var=value`**
- Use quotes to include spaces
  - **`message='Hello, world!'`**
- Refer to a variable using a dollar sign **`$`**
  - **`echo $message`**

Quoting

- Single quotes: no escapes whatsoever
  - **`'whatever i want \$"!@'`**
- Single quotes with dollar: C-style escapes
  - **`$'\'\r\n\t\a'`**
- Double quotes: allows use of **`$`**, **`` ` ``**, and **`\`**
  - **`"I can use $variables here"`** (and some other stuff we'll see shortly)
  - **`\`** can only escape **`` $`\ ``** and newlines

Commands

- Run a program with zero or more arguments
- Example: **`gcc -Wall main.c -o main.o`**
  - Runs the program **`gcc`** with the arguments **`-Wall`**, **`main.c`**, **`-o`**, and **`main.o`**
  - The shell *expands* arguments (substitutes variables, removes quotes, etc) before passing them to the program
  - The program (**`gcc`**) is then responsible for parsing arguments

Exit code

When a program ends, it returns an *exit code* – an 8-bit integer (0 to 255). 0 indicates success, nonzero codes indicate failure.

```
$ gcc a.c -o a.o
$ echo $?
0
$ gcc b.c -o b.o
cc1: fatal error: b.c: No such file or directory
compilation terminated.
$ echo $?
1
```

Standard streams

- All programs in *nix start out with three ”files” open
  - 0 – **`stdin`** (standard input)
  - 1 – **`stdout`** (standard output)
  - 2 – **`stderr`** (standard error output)
- Interactive programs take input using **`stdin`** and print output on **`stdout`**
- **`stderr`** is used for error or logging messages

Redirects and pipes

- The shell lets you control where these streams point when a command is run
- **`< file`** – replace the input stream with **`file`**
  - **`n< file`** replaces stream **`n`**
- **`> file`** – replace the output stream with **`file`**
  - **`n> file`** replaces stream **`n`** (eg. **`2> log`**)
- **`sort <words.txt >words_sorted.txt`**
- **`<words.txt >words_sorted.txt sort`**

Redirects and pipes

- While **`>`** overwrites a file, **`>>`** appends to it
- File descriptors can be duplicated to each other
  - **`n<&m`** makes **`n`** (**`stdin`** by default) an input copy of **`m`**
  - **`n>&m`** makes **`n`** (**`stdout`** by default) an output copy of **`m`**

Redirects and pipes

- Use **|** (a pipe) to connect the standard output of one command to the standard input of another
- **`<words.txt sort | head -n 1`**

And, Or, and grouping

- **`a && b`** runs **`a`** first and, if it succeeds, runs **`b`**
- **`a || b`** runs **`a`** first and, if it **`fails`**, runs **`b`**
  - The exit code of a chain of commands is the exit code of the last command in the chain
- Braces **`{ …; }`** (with semicolon!) can be used to group commands
- Parentheses **`( … )`** run a group of commands in a *subshell*

More on variables

- You can pass variables to a program by setting them as part of a command
- **`GALLIUM_DRIVER=zink glxgears`**
- Use **`export`** to mark variables as exported so children can see them
- **`export GALLIUM_DRIVER=zink`**
  `glxgears`

More on variables

Some special variables:

- **`$HOME`**: the user's home directory
- **`$IFS`**: characters used for field splitting
- **`$PATH`**: colon-separated list of directories to search for programs in
- **`$PS1`**: the prompt
- **`$PWD`**: the current working directory

More on variables

Some specialer variables:

- **`$1`**, **`$2`**, **`$3`**, etc.: the 1st, 2nd, 3rd, etc. arguments
- **`$@`** and **`$*`**: all argument passed to a script
- **`$#`**: argument count
- **`$?`**: exit code of previous command
- **`$$`**: PID of the shell
- **`$!`**: PID of the previous command

Job control

- Commands separated by **`&`** run simultaneously
  - Commands followed by **`&`** run in the background
- **`Ctrl+C`** sends SIGINT to the foreground process (usually killing it)
- **`Ctrl+Z`** sends SIGTSTP to the foreground process (usually pausing it) and moves the process to the background
- **`fg`** resumes a job and moves it to the foreground
- **`bg`** resumes a job and moves it to the background
- **`jobs`** lists current jobs

Expansion – tilde expansion

- Six expansion steps

Tilde expansion. In words beginning with a tilde

- **`~user`** expands to the home directory of **`user`**
- **`~`** expands to the current user’s home directory

Expansion – parameter expansion

- Variables and parameters (**`$example`** or **`${example}`**) are replaced with their contents
- **`${var:-default}`**: use a default value
- **`${var:=default}`**: assign a default value
- **`${var:?message}`**: create an error if **`$var`** is empty
- **`${var:+alternate}`**: substitute **`alternate`** if **`$var`** is nonempty, otherwise empty
- **`${#var}`**: string length
- **`${var%suffix}`** and **`${var%%suffix}`**: remove suffix
- **`${var#prefix}`** and **`${var##prefix}`**: remove prefix

Expansion – command substitution

- **`$(command)`** or **`` `command` ``**: execute **`command`** and use the standard output

Expansion – arithmetic

`$((expression))`

- Similar operators as in C are available

Expansion – field splitting

Previous expansions outside of double-quotes are split into multiple fields according to the value of **$IFS**

```
$ word=boysenberry
$ echo $word
boysenberry
$ IFS=e
$ echo $word
boys nb rry
```

Expansion – glob expansion

Globs are expanded to matching paths in the filesystem

1. `?` matches any character
2. `*` matches any number of characters
3. `[…]` matches any character contained within the brackets
4. `[!…]` negates this

Expansion – quote removal

Finally, quotes are removed from quoted strings.

Control flow

```
for var in a b c d
do
    echo $var
done

while command
do
    something
done

until command
do
    something
done
```

Control flow

```
if command
then
    something
elif other_command
then
    something_else
else
    third_thing
fi
```

Control flow

```
case main.c in
    *.py)
        echo "Python␣file"
        ;;
    *.c)
        echo "C␣file"
        ;;
    *)
        echo "Other"
        ;;
esac
```

Functions

Define a function (executed like a command)

```
example() echo "Hello,␣$1"
example RITlug
```