Bash Basics 4: Interactive Use

These slides introduce the basics of working with **Bash**:

1. Intro & Commands
2. Expansions & Metachars
3. Variables & Quoting
4. **Interactive Use**
   - **features specifically for interactive use**
   - **aliases and functions**
   - **startup files (to make changes permanent)**
   - **terminal control characters**
5. Redirections & Locale

Features for Interactive Use

Bash has a number of features designed specifically to enhance user productivity for *interactive use*:

- **command-line editing**
- **tab completion**
- **command history**
- **prompt** customization

Command-Line Editing

The current command line can be *edited*:
(including one retrieved from the history)

- $\langle backspace \rangle$ deletes character to left of cursor
- $\langle delete \rangle$ and ctrl-d delete character to right of cursor
- left and right arrow keys move cursor
- ctrl-a moves to start of line
- ctrl-e moves to end of line
- alt-$\langle backspace \rangle$ deletes word to left of cursor
- alt-$\langle delete \rangle$ deletes word to right of cursor
- ctrl-u erases command-line to left of cursor
- ctrl-k erases command-line to right of cursor

Tab Completion

When the user types a $\langle tab \rangle$ on the command line, Bash will try to complete the current word, as:

- a variable, if starts with $
- a username, if starts with ~
- a command, alias, or function
- a filename/directory (if no other matches are found)

If the current word is ambiguous, Bash will display the set of matches (or beep if there are no matches).

Command History

Bash maintains a **history** of the recent commands that have been executed, and you can view and recall these commands to avoid retyping them.

The `history` command lists the entire history.

The *up arrow* moves backward through history, while *down arrow* moves forward.

You can also *search* through the history:

- ctrl-r does a reverse search
- $\langle escape \rangle$ terminates the search
- ctrl-g cancels the search

The Prompt

The Bash **prompt** is controlled by **shell variable** `PS1`, and is highly customizable.

Examples:

- `PS1="> "` prompt: `>`
- `PS1="\w> "` prompt like: `~/courses/306>`
- `PS1="\h:\w> "` prompt like: `pc01:~/courses/306>`

The *default prompt* string is: `"\s-\v\$ "` (Bash version + "$ ")

See the *Bash Prompt HOWTO* for complete details and options.

Aliases

Aliases are a type of **macro** facility in the shell:
"Aliases allow a string to be substituted for a word when it is used as the first word of a simple command."

Aliases are applied as part of tokenizing the command line.

An alias is defined with the `alias` command:

```
alias ll="ls -l"
```

The `alias` command (with no arguments) will list all the currently defined aliases.

By default, aliases are applied only if the shell is *interactive*.

Aliases (contd.)

Aliases allow users to:

- Use certain command options by default:
  - `alias rm="rm -i"`
  - `alias ls="ls -F"`
- Give special/short names to commands with commonly used options and/or arguments:
  - `alias ll="ls -l"`
  - `alias up="cd .."`
- Give "better" names to standard commands:
  - `alias dir="ls"`
  - `alias filesize="du -s"`

Aliases (contd.)

Alias definitions must be placed in a Bash startup file in order to be permanently available (see below).

Many users have a separate file of alias definitions that is loaded each time they login.

Most distributions also define various aliases in the system-wide Bash startup file.

The `unalias` command can be used to disable an alias.

Functions

In Bash, aliases cannot take arguments (as they can in csh), so **functions** may have to be used instead.

A function can be defined using any of the following syntax variations:

- `function` *name* `() {` *...body...* `}`
- `function` *name* `{` *...body...* `}`
- *name* `() {` *...body...* `}`

Functions are called and executed just like simple commands.

Inside the function body, arguments to the function call can be accessed using the parameters $1, $2, etc.

Functions (contd.)

Simple example function:

```
function dos2unix ()
{
  tr -d "\r" < "$1" > "$2"
}
```

Called as:

```
dos2unix file.dos file.unix
```

Cannot be done as an alias, since “>” must be inserted into the middle of the file argument list.

Functions (contd.)

Since functions can contain arbitrary shell code, they can be much more complex than aliases:

```
#Compile C source file using GCC:
function comp ()
{
  if [[ ($# -lt 1) || ($# -gt 2) ]]; then
    echo "Usage: comp C_SOURCE_FILE [OBJECT_FILE]"
    return 1
  fi

  if [[ $# == 1 ]]; then
    gcc -Wall -std=gnu99 -o "${1%.*}" "$1"
  else
    gcc -Wall -std=gnu99 -o "$2" "$1"
  fi
}
```

Functions (contd.)

Defined like aliases at shell startup, functions like this provide an alternative to having large numbers of small shell script files.

You can check whether "*name*" names a function by using the command `type`:

- `type -t` *name*
- `type` *name* (shows the function definition)

Functions can also be used as **subroutines** in shell scripts, to name a sequence of repeatedly called shell code statements.

Startup Files

Bash reads from one or more **initialization files** when it starts.

Customization (e.g., setting the prompt, defining aliases) must be done via these files in order to be *permanent*.

If invoked as a "**login shell**," reads:

- `/etc/profile`
- followed by first found of:
  `~/.bash_profile` or `~/.bash_login` or `~/.profile`

If not a login shell, reads:

- `~/.bashrc`

User customizations typically go in `~/.bashrc`

Special Terminal Control Characters

Though actually a **terminal driver** rather than shell feature, there are *three special characters* that are frequently used when working interactively with shells:

- **ctrl-c** – sends an "**interrupt signal**" (`SIGINT`) that will usually *terminate* a running command
- **ctrl-** – sends a "**quit signal**" (`SIGQUIT`) that will usually terminate a running command
- **ctrl-d** – causes an **end of file** return when a program is reading from **standard input** (`STDIN`)