The Command Line

Matthew Bender

CMSC Command Line Workshop

Friday 18th September, 2015

Section 1

Shells

What is a shell?

What is a shell?
A shell is just another program.
It reads in a line of text from the user.
Then it processes any special characters.
The shell determines if you told it do something it already knows how to do.
If so, take that action.
Else, look for a program installed on the computer of the same name, and run that.
The shell will stop running when you tell it to (either by typing 'exit' or hitting ^D)

Example shells

- `sh` - Thompson shell - by Ken Thompson (co-creator of Unix) - first Unix shell. Very basic, 1971

Example shells

- `sh` - Thompson shell - by Ken Thompson (co-creator of Unix) - first Unix shell. Very basic, 1971
- `sh` - Bourne shell - by Stephen Bourne at Bell Labs in 1977, included scripting

Example shells

- `sh` - Thompson shell - by Ken Thompson (co-creator of Unix) - first Unix shell. Very basic, 1971
- `sh` - Bourne shell - by Stephen Bourne at Bell Labs in 1977, included scripting
- `csh` - C shell - by Bill Joy (author of the Vi editor) in 1978 - closer to C syntax than Bourne shell

Example shells

- `sh` - Thompson shell - by Ken Thompson (co-creator of Unix) - first Unix shell. Very basic, 1971
- `sh` - Bourne shell - by Stephen Bourne at Bell Labs in 1977, included scripting
- `csh` - C shell - by Bill Joy (author of the Vi editor) in 1978 - closer to C syntax than Bourne shell
- `tcsh` - improved C shell - in 1983 - offered various features like command completion

Example shells

- `sh` - Thompson shell - by Ken Thompson (co-creator of Unix) - first Unix shell. Very basic, 1971
- `sh` - Bourne shell - by Stephen Bourne at Bell Labs in 1977, included scripting
- `csh` - C shell - by Bill Joy (author of the Vi editor) in 1978 - closer to C syntax than Bourne shell
- `tcsh` - improved C shell - in 1983 - offered various features like command completion
- `bash` - Bourne-again shell - by Brian Fox in 1989. Gathered features from many shells, widely used today

Example shells

- `sh` - Thompson shell - by Ken Thompson (co-creator of Unix) - first Unix shell. Very basic, 1971
- `sh` - Bourne shell - by Stephen Bourne at Bell Labs in 1977, included scripting
- `csh` - C shell - by Bill Joy (author of the Vi editor) in 1978 - closer to C syntax than Bourne shell
- `tcsh` - improved C shell - in 1983 - offered various features like command completion
- `bash` - Bourne-again shell - by Brian Fox in 1989. Gathered features from many shells, widely used today
- `zsh` - Z shell - Paul Falstad in 1990 - Very powerful, huge set of feautures, fully customizable

Example shells

- `sh` - Thompson shell - by Ken Thompson (co-creator of Unix) - first Unix shell. Very basic, 1971
- `sh` - Bourne shell - by Stephen Bourne at Bell Labs in 1977, included scripting
- `csh` - C shell - by Bill Joy (author of the Vi editor) in 1978 - closer to C syntax than Bourne shell
- `tcsh` - improved C shell - in 1983 - offered various features like command completion
- `bash` - Bourne-again shell - by Brian Fox in 1989. Gathered features from many shells, widely used today
- `zsh` - Z shell - Paul Falstad in 1990 - Very powerful, huge set of feautures, fully customizable
- http://en.wikipedia.org/wiki/Comparison_of_command_shells

bash

`bash` is one of the most-used shells, and we will be talking about it.
Many of the features are also available in other shells, but the syntax may differ.
Bash should come installed on most systems, though it may not be the default shell.

Help in `bash`

`bash` interprets many commands on its own, which generally do not have their own `man` page.
To get help on them, you could run `man bash` and search for your desired command.
Or, use `bash`'s `help` command, which prints help about `bash`'s internal commands.

```
> help comm
```

echo

`echo` is a command that prints its arguments back to you.
It is often both an installed program, and a shell built-in, so the shell's version will run.
It is useful for seeing the values of variables and passing text to other programs.
By default, it does not interpret escape sequences. Adding the `-e` flag enables this (this must be the first argument to `echo`)

```
> echo "line 1\nline 2"
line 1\nline 2
> echo -e "line 1\nline 2"
line 1
line 2
```

Getting `bash`

Check if your default shell is `bash` - you can do this by running
`> echo $SHELL`
This tells your shell to print the default shell.
On Grace, the default shell is `tcsh`.
Because shells are just programs, you can run a shell from within another shell.
Run the command `> echo $0` to see what shell you're currently running (if it starts with a -, that means it is a login shell).
Changing shells does not change the value of the SHELL variable - the default login shell.
The file `/etc/shells` contains a list of available shells.

Changing shells example

```
> echo $SHELL
/bin/tcsh
> echo $0
-tcsh
> bash
> echo $SHELL
/bin/tcsh
> echo $0
bash
> exit
> echo $SHELL
/bin/tcsh
> echo $0
-tcsh
```

The `chsh` command

If you want to change your default shell, you can use `chsh` (**ch**ange **sh**ell).
Run `> chsh -s shellname` to change your login shell (`shellname` must be the full path to the shell, like `/bin/bash`).
You can a list of available shells with `cat /etc/shells`.
If you're on Grace, `chsh` is disabled (in a way) for reasons I am still unsure about. I suggest just running `bash` when you login.

Section 2

Shell features

Variables

`bash` treats everything as a string.
Assigning to a variable: `> name=Matthew`
Use $ to get back the variable:

```
> echo My name is $name
My name is Matthew
```

If the variable is not set, the result is the empty string.
Many environment variables are all caps, so it is recommended you use lowercase.

Variables

Strings can be combined by putting them next to each other.

```
> a=fruit
> b=fly
> echo $a$b
fruitfly
> echo butter$b
butterfly
> echo $as
```

This will print nothing, as the variable `as` is not set.

How do we print `fruits` then?

Variables

We can be more explicit with our variables:

```
> echo ${a}s
fruits
```

We could also quote the variable:

```
> echo "$a"s
fruits
```

The `read` command

Use `read` to read in a variable.
Lets the user enter a string into the variable `$var`:

```
> read var
```

Add a prompt string:

```
> read -p "Enter username:  " username
```

Don't have the text show up when the user types:

```
> read -s -p "Enter password:  " password
```

Environment Variables

Use the `env` command to see all the environment variables.

- `$HOME` = your home directory (~ also evaluates to this)

Environment Variables

Use the `env` command to see all the environment variables.

- `$HOME` = your home directory (`~` also evaluates to this)
- `$PATH` = a colon-separated list of directories that the shell looks for programs to run

Environment Variables

Use the `env` command to see all the environment variables.

- `$HOME` = your home directory (`~` also evaluates to this)
- `$PATH` = a colon-separated list of directories that the shell looks for programs to run
- `$PAGER` = default pager (e.g. `less`, `more` used to view text when there is too much to fit on the screen)

The `$PATH` variable

The `$PATH` variable is a colon-separated list of directories.
When your shell doesn't know how to do something (e.g. `mkdir`), it searches the directories listed in `$PATH`, one by one.
The directories of the `$PATH` variable will be searched, one-by-one, until one containing the specified program is found. Then that one is run.
If running `> echo $PATH` isn't readable enough for you, try
`> echo $PATH | tr : '\n'`.
We will cover exactly how this works later, but as a short explanation, it send the output of `echo $PATH` to the input of `tr : '\n'`, which will translate colons to newlines.

Wildcards

Special characters that expand based on files in the given directory.
Also called "globs".
`*` - expands to any sequence of characters
`?` - expands to any 1 character
See `man 7 glob` for more information.

Wildcard examples

```
> ls
a b band.txt bend.c bend.txt
> echo *
a b band.txt bend.c bend.txt
> echo ?
a b
> echo bend.*
bend.c bend.txt
> echo b?nd.txt
band.txt bend.txt
> echo b?nd.*
band.txt bend.c bend.txt
> echo b*
b band.txt bend.c bend.txt
```

Hidden Files

Files and directories that start with a `.` are "hidden".
These are often used for configuration files and directories.
These will not show up in a `ls`, but you can use `ls -a` to show them.
The wildcard `*` will not match them - use `.*` instead.
Use both to match all files in a directory:
`> echo * .*` will show all files (like `ls -a`)

Aliases

Simple abbreviations for more complex commands.

`alias abbreviation=longer_command`

If `longer_command` contains spaces, you have to quote it.

```
> alias lsa="ls -a"
> lsa
.  ..  .dotfile file1 file2
```

You can also alias commands that already exist. A very common alias:

```
> alias ls="ls --color"
```

Typing `alias` without any arguments will print all the current aliases.

Typing `alias comm` will print whatever `comm` is currently aliased to.

If you alias over a command, put a backslash first to run the unaliased command: `> \ls` runs the original version of `ls`

Use `unalias` to unalias a command.

Common Aliases

When removing files, ask before deleting each.

```
> alias rm="rm -i"
```

When moving files, ask for confirmation if doing so would overwrite another file.

```
> alias mv="mv -i"
```

When copying files, ask for confirmation if doing so would overwrite another file.

```
> alias cp="cp -i"
```

Move to parent directory, and move to parent directory's parent directory

```
> alias ..="cd .."
> alias ...="cd ../.."
```

Exit codes

Commands have an exit code.
0 means success, anything non-zero means failure.
In some code you write, you can specify this when you exit (`exit(exit_code);` in C, or the value you return from `main`).
In `bash`, the `$?` variable contains the exit code of the last command.
`true` is a command that does nothing and exits with code 0, `false` does nothing and exits with code 1.
Most command will set a non-zero exit code if you tell it to do something it can't (like remove a non-existant file).

;, && and ||

> `comm1; comm2` will run `comm1`, followed by `comm2`
> `comm1 && comm2` will run `comm1`, and then if successful (exit code of 0), also runs `comm2`
> `comm1 || comm2` will run `comm1`, and then if unsuccessful (exit code not 0), also runs `comm2`

We can use this to print if a command succeeds or fails:

```
> comm && echo Success || echo Failure
```

IO Redirection

Programs have 3 basic IO (input/output) streams: `stdin` (standard input), `stdout` (standard output), and `stderr` (standard error)
If a program requests input on `stdin`, it will wait until you enter text at the keyboard (generally the terminal buffers this until you enter a newline).
If a program writes something to `stdout` or `stderr`, this will be printed out to you.
In C, use `printf("Output")` or `fprintf(stdout, "Output")` to print to `stdout`.
Use `fprintf(stderr, "Error")` to print to `stderr`.

Output Redirection

`$ comm > file` will run `comm`, and redirect stdout to the file `file` instead of printing it to the screen. Any previous contents of `file` are erased. (Depending on `bash`'s settings, it may stop you from overwriting files like this. Run `set +o noclobber` to disable this.)
`$ comm >> file` will run `comm`, and redirect stdout to append to the end of `file` instead of printing it to the screen. The previous contents of `file` are preserved.
The above command only redirects `stdout`, not `stderr`. To redirect `stderr`, put a `2` in front, like `$ comm 2> file` or `$ comm 2>> file`.
If a program prints to both `stdout` and `stderr`, you can separate these by redirecting one of them and leaving the other to print to the screen, or write both to separate files:
`$ comm > outfile 2> errfile`

Output Redirection

You can also redirect `stdout` to wherever `stderr` is, or vice-versa like so:

```
$ comm 2>&1 # redirects stderr to stdout
$ comm >&2 # redirects stdout to stderr
```

Note that `stdout` is file descriptor 1, and `stderr` is file descriptor 2, hence the syntax. You can omit the 1 when redirecting `stdout`.

To ignore the output from a command, redirect it to the special file `/dev/null`.

For example, the following command removes all output by first redirecting `stdout` to `/dev/null`, and then redirecting `stderr` to wherever `stdout` is going:

```
$ comm > /dev/null 2>&1
```

Input Redirection

If a program reads from `stdin`, you can use the contents of a file instead like so:

```
$ comm < file # reads input from file as if it was stdin

$ comm << TOKEN # reads stdin until TOKEN is read
input line1
input line2
input line3
TOKEN # TOKEN read, input stops
```

`TOKEN` can be any word you want.

This is useful because you don't have to create a file to read from, and it allows you to repeat the command and edit it without re-entering the text.

If a program is reading from `stdin` and you are entering text, hit ^D to signal end of `stdin`.

Piping

A pipe redirects the `stdout` from one command to the `stdin` of another. One of the most useful tools you have - it allows you to combine several small, simple programs into a tool suited for your needs.

```
$ cat animals.txt | fgrep -i monkey | wc -l
```

This sends the contents of `animals.txt` to the `fgrep` program, which outputs any lines that match "monkey", ignoring case (the `-i` flag). Finally, that output is send to the `wc` program (**w**ord **c**ount), which prints the number of lines (the `-l` flag).

We just combined small commands to count the number of monkeys in our file.

Piping to a pager

When the output of a command is too long, it helps to pipe the output to a pager, such as `less` or `more`.
Pagers allow you to scroll through text.
In `less`, use arrow keys or PgUp/PgDn to scroll. Hit 'q' to exit. Hit 'h' for help, and 'q' to exit the help.
Use '/abc' to search for text containing 'abc', and 'n' and 'N' to move forward and back among the matches.
'g' will take you to the top, and 'G' to the bottom.
Pipe the output of a command to less to scroll through all of it:

```
> comm | less
```

The `tee` command

Sometimes it is helpful to both see the output of a command and save it to a file.
`tee` copies its `stdin` to a file, and also prints it to `stdout`
`$ comm | tee file` will save the output of `comm` in `file`, and also print it to `stdout`.
Use `tee -a` to append to the file instead of overwriting it.

Grouping

You may want to group several command together, like parentheses in other programming languages.
The syntax for this is to surround them with {}'s, and leave a semicolon after the last command. There must be a space after the first {.
Compare:

```
$ echo one; echo two > nums.txt # prints "one" and writes "two" to nums.txt
$ { echo one; echo two; } > nums.txt # prints nothing, writes "one" and "two" to nums.txt
```

Arguments and Quoting

1) `bash` reads a line of text from the user
2) `bash` processes the line for special characters like `$`, `*`, etc. and translates these accordingly.
`rm $prog1/*.o` might translate to
`rm /homes/bender/216/projects/1/list.o /homes/bender/216/projects/1/test.o`
3) `bash` then splits this on whitespace into a list of tokens:
`[rm, /homes/bender/216/projects/1/list.o, /homes/bender/216/projects/1/test.o]`
4) `bash` then calls the program given by the first token, with the arguments given by the remaining arguments
Note that programs do not see the special characters themselves, `bash` translates them first.

Arguments in programs

Programming languages have a feature to accept arguments.
Java: `public static void main(String[] args)`
`args` is an array of Strings containing the arguments.
C: `int main(int argc char * argv[])`
`argv` is an array of character pointers containing the arguments, `argc` is the number of arguments.
Python: `import sys`
`sys.argv` is a list of the arguments
In `bash`, aliases take no arguments, they just expand, and you can pass arguments to what they expand to only on the end.
`bash` functions and shell scripts both take arguments.
Note that some languages will not include the executable as an argument.

Printing Program Arguments Example

```
#include <stdio.h>

int main(int argc, char * argv[]) {
 int i;
 for (i = 0; i < argc; i++) {
    printf("%d:  %s\n", i, argv[i]);
 }
 return 0;
}
```

Save this as `argprint.c`, run the following:

```
> gcc argprint.c -o argprint
```

This produces an executable called `argprint`.

argprint

```
> argprint
0:  argprint
> argprint ONE TWO
0:  argprint
1:  ONE
2:  TWO
> argprint *.txt
0:  argprint
1:  emails.txt
2:  stuff.txt
3:  test.txt
> argprint ~ .
0:  argprint
1:  /homes/bender
2:  .
```

Configuration Files

Using commands like `alias la="ls -a"` or `set +o noclobber` only last for the duration of the shell.
Many programs have configuration files that they read on startup to determine settings.
Most of the time, these are start with a `.` and end with `rc` (the `rc` stands for **r**un **c**ommands).
Example: when `bash` starts up, it reads from the `.bashrc` file.
If you want to print a greeting everytime you start bash, add `echo Hello!` to the end of your `.bashrc` file.
On Grace, the `.bashrc` file is read-only, but you can edit the `.bashrc.mine` file.
So add your options like `set +o noclobber` to your `.bashrc` or `.bashrc.mine` file.
Note: all these config files are in your home directory.

Sourcing Files

In `bash`, you can run `source file` to run the commands in the current shell. So you can `source` a file containing options or aliases to have them loaded in the current shell.
For example, `bash` `sources` your `.bashrc` file on startup. This in turn `sources` your `.bash_aliases` file, so put your aliases there (they would work just as well in `.bashrc` though).
A synonym for `source` is `.`, so `source .bashrc` and `. .bashrc` do the same thing.
If you edit your `.bashrc` or `.bash_aliases`, you will have to `source` them or logout and log back in to get the changes.

Want to be evil?

If you really want to be evil, and you have access to your friend's `.bashrc`, add the following line:

```
echo "sleep 0.5" >> ~/.bashrc
```

This cause the line `sleep 0.5` to be added to your friend's `.bashrc` everytime he logs in.

This means his login time will appear to slowly increase everytime he logs in, because there will be more and more lines of `sleep 0.5` in his `.bashrc`

Useful sourcing aliases

In my `.bash_aliases`, I like to have the following 2 lines:

```
alias bashrc="vim ~/.bashrc && source ~/.bashrc"
alias als="vim ~/.bash_aliases && source ~/.bash_aliases"
```

These will open up my `.bashrc` or `.bash_aliases` file, respectively, and `source` them when I am done editing them so I don't have to.

Command History

The `history` command prints your history of commands entered.
`!!` is the last command you entered.
`!n` is the nth command in your history.
`!-n` is the nth from last command in your history.
`!abc` is the last command starting with `abc`.
`!?abc` is the last command containing `abc`.
`!*` is all of the arguments to the last command.
`!^` is the first argument of the last command.
`!$` is the last argument of the last command.
`!!:4` is the 4th argument of the last command (also works with `!-2:4`, `!v:4`, etc.).
`!!:2-5` is the 2nd through 5th arguments of the last command (also works with other command like above).

Shell Builtins

As mentioned before, there are some things `bash` knows how to do. For everything else, ~~there's Mastercard~~ it looks for an installed program of the given name.
Some examples of `bash` builtins:
`alias` - create an alias for a command
`function` - create a function (more powerful than an alias)
`echo` - `bash` provides its own implementation of echo
`shopt` - set or unset various shell options
`type` - see what type of command something is

Shell Builtins

To what `bash` thinks your command is, use the `type` command:

```
> type comm
```

This will print one of several things:

- "`comm is a shell builtin`" if `comm` is one of `bash`'s internal commands.

Shell Builtins

To what `bash` thinks your command is, use the `type` command:

```
> type comm
```

This will print one of several things:

- "`comm is a shell builtin`" if `comm` is one of `bash`'s internal commands.
- "`comm is aliased to 'blah blah blah'`" if `comm` is a user-defined alias.

Shell Builtins

To what `bash` thinks your command is, use the `type` command:

```
> type comm
```

This will print one of several things:

- "`comm is a shell builtin`" if `comm` is one of `bash`'s internal commands.
- "`comm is aliased to 'blah blah blah'`" if `comm` is a user-defined alias.
- "`comm is a function`", followed by its implementation, if `comm` is a user-defined function.

Shell Builtins

To what `bash` thinks your command is, use the `type` command:

```
> type comm
```

This will print one of several things:

- "`comm is a shell builtin`" if `comm` is one of `bash`'s internal commands.
- "`comm is aliased to 'blah blah blah'`" if `comm` is a user-defined alias.
- "`comm is a function`", followed by its implementation, if `comm` is a user-defined function.
- "`comm is /bin/comm`", or some other path, if `comm` is a installed program.

Shell Builtins

To what `bash` thinks your command is, use the `type` command:

```
> type comm
```

This will print one of several things:

- "`comm is a shell builtin`" if `comm` is one of `bash`'s internal commands.
- "`comm is aliased to 'blah blah blah'`" if `comm` is a user-defined alias.
- "`comm is a function`", followed by its implementation, if `comm` is a user-defined function.
- "`comm is /bin/comm`", or some other path, if `comm` is a installed program.
- "`comm is hashed (/bin/comm)`", if `comm` is a installed program and has been recently used. `bash` remembers where is is located to avoid having to re-search for it in `$PATH`. Unhash it with `unhash comm`.

Shell Builtins

To what `bash` thinks your command is, use the `type` command:

```
> type comm
```

This will print one of several things:

- "`comm is a shell builtin`" if `comm` is one of `bash`'s internal commands.
- "`comm is aliased to 'blah blah blah'`" if `comm` is a user-defined alias.
- "`comm is a function`", followed by its implementation, if `comm` is a user-defined function.
- "`comm is /bin/comm`", or some other path, if `comm` is a installed program.
- "`comm is hashed (/bin/comm)`", if `comm` is a installed program and has been recently used. `bash` remembers where is is located to avoid having to re-search for it in `$PATH`. Unhash it with `unhash comm`.
- "`comm` is a shell keyword" if `comm` is a keyword like `if` or `!`

The `which` command

`which command` will find the first instance of `command` in your path; that is, which executable program would be run if you had run `command` Add the `-a` flag to print all the executables matching `command` in your path.

Quoting

In `bash`, everything is separated by whitespace
You cannot run `name=Matthew Bender`, because `Matthew Bender` is not seen as one string.
Similarly, running `> mkdir Project 1` will create two directories, `Project` and `1`
To fix this, you can escape whitespace with a backslash:

```
> mkdir Project\ 1
```

Or add quotes:

```
> mkdir "Project 1"
> mkdir 'Project 1'
```

The difference is evident if we pass these to our `argprint` program from earlier.

Quoting

Be careful using variables with whitespace: they will expand into multiple arguments. Quote them to resolve this:

```
> proj="Project 1"
> cd $proj # will not work, cd receives 2 arguments: Project, and 1
> cd "$proj" # will work, as bash treats quoting strings as 1 argument
> cd '$proj' # will not work, variables are not expanded in single quotes
```

This is why it is generally best practice to quote your variables.

Brace Expansions

A comma-separated list inside braces will be expanded to each of its arguments, with any text around them being added around each argument.

```
mkdir 216/{projects,homeworks,notes}
```

Expands to:

```
mkdir 216/projects 216/homeworks 216/notes
```

```
echo "a "{man,plan,canal}, Panama
```

Prints:

```
a man, a plan, a canal, Panama
```

```
cp path/to/file{,.bak} # copies path/to/file to
path/to/file.bak
```

Brace Expansions

Two numbers or charactes separated by 2 dots in braces expand to the given range.

```
mkdir 216/projects/p{1..5}
```

Expands to:

```
mkdir 216/projects/p1 216/projects/p2
216/projects/p3 216/projects/p4 216/projects/p5
```

```
echo {A..E} {f..j}
```

Prints:

```
A B C D E f g h i j
```

Ranges can take a step as well:

```
echo {A..Z..3}
```

Prints ever 3rd letter:

```
A D G J M P S V Y
```

Brace Expansions

You can combine multiple brace expansions in the same expression.

`alias chess_squares="echo {a..h}-{1..8}"`
`chess_squares` prints:

```
a-1 a-2 a-3 a-4 a-5 a-6 a-7 a-8 b-1 b-2 b-3 b-4 b-5
b-6 b-7 b-8 c-1 c-2 c-3 c-4 c-5 c-6 c-7 c-8 d-1 d-2
d-3 d-4 d-5 d-6 d-7 d-8 e-1 e-2 e-3 e-4 e-5 e-6 e-7
e-8 f-1 f-2 f-3 f-4 f-5 f-6 f-7 f-8 g-1 g-2 g-3 g-4
g-5 g-6 g-7 g-8 h-1 h-2 h-3 h-4 h-5 h-6 h-7 h-8
```

Brace Expansions

```
for i in {5..1}; do
 echo "$i"...
 sleep 1
done
echo Blastoff!
```

Command Substitution

`bash` provides syntax to substitute the output of a command as other arguments.
2 different notations are used: surround the command in backticks, or surround it with `$()`. The latter notation is preferred, mainly because it allows easy nesting.

```
> rm $(cat files_to_delete.txt) # removes the files listed in files_to_delete.txt
> now=$(date) # sets the $now variable to be the current date and time
> vim $(ls | fgrep -i "list") # opens all files in the current directory with "list" in their name, ignoring case
```