The Command Line

Matthew Bender

CMSC Command Line Workshop

Friday 9th October, 2015

Section 1

Functions

Combining Commands

Often you will find yourself repeating a series of commands.
You can combine these into either a function or a shell script.
These can take arguments on the command line just like any other command.

Functions

```
function funcname1 {
 comm1
 comm2
}
```

Or (arguments do not go in the parentheses):

```
funcname2 () {
 comm1
 comm2
}
```

Also on 1 line (note the semicolon before the closing brace):

```
function funcname3 { comm1; comm2; }
```

Function Example

```
function setup {
 cd ~/216/projects/1
 clear
 date
 echo Welcome back
}
```

Call functions like any other command (no parentheses):

```
> setup
```

Function arguments

Function arguments are stored as variables `$1`, `$2`, etc.
After the 9th argument, you must use braces: `${10}`, `${11}`, etc.
The `$@` variable holds all the arguments passed.
The `$#` arguments holds the number of arguments passed.
The `$0` variable still holds the name of the shell you're running.
The `$FUNCNAME` variable holds the name of the function.
Remember to quote these variables or they will not work when passing arguments with whitespace.

Function arguments

```
function funcinfo {
 echo Function:  "$FUNCNAME"
 echo \# args:  $#
 echo arg 1:  "$1"
 echo arg 2:  "$2"
 echo all args:  "$@"
}
> funcinfo foo bar # call like regular command

function mkdircd {
 mkdir "$1" && cd "$1"
}
```

What happens if we pass 0 arguments to `mkdircd`?

Section 2

Shell Scripting

Shell Scripts

Shell scripts are another way to group commands to create programs
Save them in a file, usually with a `.sh` extension (though not necessary)
When ready, run `> bash script.sh` to run your script
This launches a new instance of `bash`, which runs the commands in `script.sh` instead of commands entered from the user

Shell Scripts

You can also execute the script directly using a shebang.
A shebang is a character sequence at the top of a file telling a shell what program to execute it with.
Put a `#!` (the shebang) at the top of your script, followed by the full path to the bash executable at the top of your script (find this with `which bash`).

```
#!/bin/bash
echo Hello, world
```

You must then make the script executable:

```
$ chmod +x script.sh
```

You can then run it by specifying the path to it as the command:

```
./script.sh
```

Or if the `.` directory is in your path, just run `> script.sh`.
(The executable after the shebang can be any interpreter, like another shell, or `python` or `perl`)

Script arguments

Positional arguments work like functions: `$1`, `$2`, etc.
The difference is that `$0` is now the script name.
And `$FUNCNAME` is not set.

Difference between functions and scripts

- Functions exist in the environment of a shell. A function you define in this shell will not exist in another instance. You can get around this by defining the function in your `.bashrc`
- Shell scripts are just files independent of the shell. You can move them around, put them in your path, copy and edit them, email them, etc.
- Functions have access to the entire environment of the shell - aliases, other functions, variables, etc.
- Shell scripts do not, because a new shell is started every time they are run.
- Use functions for small things that are more complicated than an alias could do.
- Use shell scripts for larger, more complicated things that might have to be modified and maintained.

Section 3

Control Flow

The `test` command

The `test` command is used to check some conditional. If the conditional is true, it sets its exit code to 0, else something non-zero.
`test -f blah` checks if `blah` is a regular file
`test -d blah` checks if `blah` is a directory
`test -n str` checks if the length of `str` is non-zero
`test -z str` checks if the length of `str` is zero
`test str1 = str2` checks string equality
`test str1 != str2` checks string inequality
`test int1 -eq int2` checks integer equality
Instead of `-eq`, use `-ne`, `-gt`, `-ge`, `-lt`, and `-le` for not equals, greater than, greater than or equals, less than, and less than or equals

The `test` command

`test ! expr` tests if `expr` is false
`test expr1 -a expr2` tests if both are true
`test expr1 -o expr2` tests if either are true
Instead of saying `test expr`, you can say `[ expr ]`,
e.g. `[ -f blah.txt ]`
Make sure you surround `expr` with spaces.

Control flow: `if`

```
if comm1; then
 expr1;
elif comm2; then
 expr2;
else
 expr3;
fi
```

if example

```
read -p "Who are you?  " name
if test "$name" = "Matt"; then
 echo "Hello!"
elif [ "$name" = "John" ]; then
 echo "Hey there"
else
 echo "I don't know you"
fi
```

Control flow: `while`

```
while comm; do
 expr
done
```

Example:

```
while true; do
 echo Another minute...
 sleep 60
done
```

Looping through arguments

The `shift` command puts `$2` into `$1`, `$3` into `$2`, etc. and decrements `$#`

```
while [ $# -ge 1 ]; do
 echo "$1";
 shift;
done
```

Control flow: `for`

```
for var in arg1 arg2 argN; do
 expr "$var"
done
```

Example: backing up files

```
for file in *.c *.h; do
 cp "$file" "$file".bak
done
```

`for` loop example: testing your project

```
for i in {1..6}; do
 ./project1 < public"$i".in > test"$i".out
 if cmp -s public"$i".out public"$i".out; then
  echo "public test $i succeeded"
 else
  echo "public test $i failed"
 fi
done
```

Control flow: `case`

```
case expr in
pattern1)
 comm1
 ;;
pattern2)
 comm2
 ;;
pattern3)
 comm3
 ;;
esac
```

`case` example

```
read -p "What class are you in?  " class
case "$class" in
13?)
 echo "Enjoy Java!"
 ;;
216)
 echo "C isn't so bad"
 ;;
4*)
 echo "You're in some hard classes"
 ;;
*)
 echo "I don't know that class"
 ;;
esac
```