Bash scripts

ComS 252 — Iowa State University

Andrew Miner

Motivation

- Suppose we have a sequence of commands, executed regularly
- For example, suppose we back up user files with:
  1. `mount /dev/hdb3 /mnt/backup`
  2. `cd /mnt/backup`
  3. `mv -f backup.tgz backup.old.tgz`
  4. `tar czf backup.tgz /home`
  5. `cd /`
  6. `umount /mnt/backup`
- If we put these in a text file, how can we execute the file?
- These text files are called shell scripts

But first: some useful utilities for shell scripts

`printf`

- Acts a lot like `printf` in C
- Except it deals entirely with strings
- Useful for more controlled I/O than plain old `echo`

```
prompt$
```

But first: some useful utilities for shell scripts

`printf`

- Acts a lot like `printf` in C
- Except it deals entirely with strings
- Useful for more controlled I/O than plain old `echo`

```
prompt$ printf "Hello, world.\n"
```

But first: some useful utilities for shell scripts

`printf`

- Acts a lot like `printf` in C
- Except it deals entirely with strings
- Useful for more controlled I/O than plain old `echo`

```
prompt$ printf "Hello, world.\n"
Hello, world.
prompt$
```

But first: some useful utilities for shell scripts

`printf`

- Acts a lot like `printf` in C
- Except it deals entirely with strings
- Useful for more controlled I/O than plain old `echo`

```
prompt$ printf "Hello, world.\n"
Hello, world.
prompt$ printf "Decimal 42 in hex is %x\n" "42"
```

But first: some useful utilities for shell scripts

`printf`

- Acts a lot like `printf` in C
- Except it deals entirely with strings
- Useful for more controlled I/O than plain old `echo`

```
prompt$ printf "Hello, world.\n"
Hello, world.
prompt$ printf "Decimal 42 in hex is %x\n" "42"
Decimal 42 in hex is 2a
prompt$
```

But first: some useful utilities for shell scripts

printf

- Acts a lot like `printf` in C
- Except it deals entirely with strings
- Useful for more controlled I/O than plain old `echo`

```
prompt$ printf "Hello, world.\n"
Hello, world.
prompt$ printf "Decimal 42 in hex is %x\n" "42"
Decimal 42 in hex is 2a
prompt$ printf "I deposited\n\t$%.2f in the bank.\n" 4.1
```

But first: some useful utilities for shell scripts

printf

- Acts a lot like `printf` in C
- Except it deals entirely with strings
- Useful for more controlled I/O than plain old `echo`

```
prompt$ printf "Hello, world.\n"
Hello, world.
prompt$ printf "Decimal 42 in hex is %x\n" "42"
Decimal 42 in hex is 2a
prompt$ printf "I deposited\n\t$%.2f in the bank.\n" 4.1
I deposited
        $4.10 in the bank.
prompt$
```

But first: some useful utilities for shell scripts

`mktemp`

- Creates an empty temporary file
- For example, as might be used within a shell script
- Usage: `mktemp template`
  - "`template`" is a path name containing 6 consecutive '`X`'s
  - The '`X`'s are replaced to make a unique name
  - The path name is written to standard output
- Check your `man` pages for more details

```
prompt$
```

But first: some useful utilities for shell scripts

`mktemp`

- Creates an empty temporary file
- For example, as might be used within a shell script
- Usage: `mktemp template`
  - "`template`" is a path name containing 6 consecutive '`X`'s
  - The '`X`'s are replaced to make a unique name
  - The path name is written to standard output
- Check your `man` pages for more details

```
prompt$ mktemp /tmp/myfile.XXXXXX
```

But first: some useful utilities for shell scripts

mktemp

- Creates an empty temporary file
- For example, as might be used within a shell script
- Usage: `mktemp template`
  - "`template`" is a path name containing 6 consecutive '`X`'s
  - The '`X`'s are replaced to make a unique name
  - The path name is written to standard output
- Check your `man` pages for more details

```
prompt$ mktemp /tmp/myfile.XXXXXX
/tmp/myfile.rGaX8K
prompt$
```

But first: some useful utilities for shell scripts

mktemp

- Creates an empty temporary file
- For example, as might be used within a shell script
- Usage: `mktemp template`
  - "`template`" is a path name containing 6 consecutive '`X`'s
  - The '`X`'s are replaced to make a unique name
  - The path name is written to standard output
- Check your `man` pages for more details

```
prompt$ mktemp /tmp/myfile.XXXXXX
/tmp/myfile.rGaX8K
prompt$ tempf=$(mktemp /tmp/myfile.XXXXXX)
```

But first: some useful utilities for shell scripts

mktemp

- Creates an empty temporary file
- For example, as might be used within a shell script
- Usage: `mktemp template`
  - "`template`" is a path name containing 6 consecutive '`X`'s
  - The '`X`'s are replaced to make a unique name
  - The path name is written to standard output
- Check your `man` pages for more details

```
prompt$ mktemp /tmp/myfile.XXXXXX
/tmp/myfile.rGaX8K
prompt$ tempf=$(mktemp /tmp/myfile.XXXXXX)
prompt$
```

But first: some useful utilities for shell scripts

mktemp

- Creates an empty temporary file
- For example, as might be used within a shell script
- Usage: `mktemp template`
  - "`template`" is a path name containing 6 consecutive '`X`'s
  - The '`X`'s are replaced to make a unique name
  - The path name is written to standard output
- Check your `man` pages for more details

```
prompt$ mktemp /tmp/myfile.XXXXXX
/tmp/myfile.rGaX8K
prompt$ tempf=$(mktemp /tmp/myfile.XXXXXX)
prompt$ ls -l > $tempf
```

But first: some useful utilities for shell scripts

mktemp

- Creates an empty temporary file
- For example, as might be used within a shell script
- Usage: `mktemp template`
  - "`template`" is a path name containing 6 consecutive '`X`'s
  - The '`X`'s are replaced to make a unique name
  - The path name is written to standard output
- Check your `man` pages for more details

```
/tmp/myfile.rGaX8K
prompt$ tempf=$(mktemp /tmp/myfile.XXXXXX)
prompt$ ls -l > $tempf
prompt$
```

But first: some useful utilities for shell scripts

mktemp

- Creates an empty temporary file
- For example, as might be used within a shell script
- Usage: `mktemp template`
  - "`template`" is a path name containing 6 consecutive '`X`'s
  - The '`X`'s are replaced to make a unique name
  - The path name is written to standard output
- Check your `man` pages for more details

```
/tmp/myfile.rGaX8K
prompt$ tempf=$(mktemp /tmp/myfile.XXXXXX)
prompt$ ls -l > $tempf
prompt$ ls /tmp/myfile*
```

But first: some useful utilities for shell scripts

mktemp

- Creates an empty temporary file
- For example, as might be used within a shell script
- Usage: `mktemp template`
  - "`template`" is a path name containing 6 consecutive '`X`'s
  - The '`X`'s are replaced to make a unique name
  - The path name is written to standard output
- Check your `man` pages for more details

```
prompt$ ls -l > $tempf
prompt$ ls /tmp/myfile*
/tmp/myfile.rGaX8K  /tmp/myfile.ZTK0g6
prompt$
```

But first: some useful utilities for shell scripts

mktemp

- Creates an empty temporary file
- For example, as might be used within a shell script
- Usage: `mktemp template`
  - "`template`" is a path name containing 6 consecutive '`X`'s
  - The '`X`'s are replaced to make a unique name
  - The path name is written to standard output
- Check your `man` pages for more details

```
prompt$ ls -l > $tempf
prompt$ ls /tmp/myfile*
/tmp/myfile.rGaX8K  /tmp/myfile.ZTK0g6
prompt$ rm -f $tempf
```

But first: some useful utilities for shell scripts

mktemp

- Creates an empty temporary file
- For example, as might be used within a shell script
- Usage: `mktemp template`
  - "`template`" is a path name containing 6 consecutive '`X`'s
  - The '`X`'s are replaced to make a unique name
  - The path name is written to standard output
- Check your `man` pages for more details

```
prompt$ ls /tmp/myfile*
/tmp/myfile.rGaX8K  /tmp/myfile.ZTK0g6
prompt$ rm -f $tempf
prompt$
```

Executing a script

Suppose the text file "`script`" contains shell commands.
How can we execute the script?

Executing a script

Suppose the text file "`script`" contains shell commands.
How can we execute the script?

1. Use redirection:

```
prompt$ bash < script
```

Executing a script

Suppose the text file "`script`" contains shell commands.
How can we execute the script?

1. Use redirection:

```
prompt$ bash < script
```

2. Use pipes (better for "dynamically generated" commands):

```
prompt$ cat script | bash
```

Executing a script

Suppose the text file "`script`" contains shell commands.
How can we execute the script?

1. Use redirection:

```
prompt$ bash < script
```

2. Use pipes (better for "dynamically generated" commands):

```
prompt$ cat script | bash
```

3. Specify the input file as an argument:

```
prompt$ bash script
```

Executing a script

Suppose the text file "`script`" contains shell commands.
How can we execute the script?

1. Use redirection:

```
prompt$ bash < script
```

2. Use pipes (better for "dynamically generated" commands):

```
prompt$ cat script | bash
```

3. Specify the input file as an argument:

```
prompt$ bash script
```

4. How can I make it so that I can simply type:

```
prompt$ script
```

The "Hello, world" of shell scripts

- Let's build a simple script and see how to make it run

The "Hello, world" of shell scripts

- Let's build a simple script and see how to make it run

```
prompt$ cat > hello
```

The "Hello, world" of shell scripts

- Let's build a simple script and see how to make it run

```
prompt$ cat > hello
```

The "Hello, world" of shell scripts

- Let's build a simple script and see how to make it run

```
prompt$ cat > hello
echo 'Hello, world!'
```

The "Hello, world" of shell scripts

- Let's build a simple script and see how to make it run

```
prompt$ cat > hello
echo 'Hello, world!'
```

The "Hello, world" of shell scripts

- Let's build a simple script and see how to make it run

```
prompt$ cat > hello
echo 'Hello, world!'
prompt$
```

The "Hello, world" of shell scripts

- Let's build a simple script and see how to make it run

```
prompt$ cat > hello
echo 'Hello, world!'
prompt$ bash hello
```

The "Hello, world" of shell scripts

- Let's build a simple script and see how to make it run
- Is this enough? Let's try.

```
prompt$ cat > hello
echo 'Hello, world!'
prompt$ bash hello
Hello, world!
prompt$
```

The "Hello, world" of shell scripts

- Let's build a simple script and see how to make it run
- Is this enough? Let's try.

```
prompt$ cat > hello
echo 'Hello, world!'
prompt$ bash hello
Hello, world!
prompt$ hello
```

The "Hello, world" of shell scripts

- Let's build a simple script and see how to make it run
- Is this enough? No. What happened?

```
prompt$ cat > hello
echo 'Hello, world!'
prompt$ bash hello
Hello, world!
prompt$ hello
-bash: hello: command not found
prompt$
```

The "Hello, world" of shell scripts

- Let's build a simple script and see how to make it run
- Is this enough? No. What happened?
- `hello` is not in my `PATH`

```
prompt$ cat > hello
echo 'Hello, world!'
prompt$ bash hello
Hello, world!
prompt$ hello
-bash: hello: command not found
prompt$
```

The "Hello, world" of shell scripts

- Let's build a simple script and see how to make it run

```
prompt$ cat > hello
echo 'Hello, world!'
prompt$ bash hello
Hello, world!
prompt$ hello
-bash: hello: command not found
prompt$ pwd
```

The "Hello, world" of shell scripts

- Let's build a simple script and see how to make it run

```
echo 'Hello, world!'
prompt$ bash hello
Hello, world!
prompt$ hello
-bash: hello: command not found
prompt$ pwd
/home/alice
prompt$
```

The "Hello, world" of shell scripts

- Let's build a simple script and see how to make it run

```
echo 'Hello, world!'
prompt$ bash hello
Hello, world!
prompt$ hello
-bash: hello: command not found
prompt$ pwd
/home/alice
prompt$ PATH="$PATH:/home/alice"
```

The "Hello, world" of shell scripts

- Let's build a simple script and see how to make it run
- Is this enough? Let's try.

```
prompt$ bash hello
Hello, world!
prompt$ hello
-bash: hello: command not found
prompt$ pwd
/home/alice
prompt$ PATH="$PATH:/home/alice"
prompt$
```

The "Hello, world" of shell scripts

- Let's build a simple script and see how to make it run
- Is this enough? Let's try.

```
prompt$ bash hello
Hello, world!
prompt$ hello
-bash: hello: command not found
prompt$ pwd
/home/alice
prompt$ PATH="$PATH:/home/alice"
prompt$ hello
```

The "Hello, world" of shell scripts

- Let's build a simple script and see how to make it run
- Is this enough? No. What happened?

```
prompt$ hello
-bash: hello: command not found
prompt$ pwd
/home/alice
prompt$ PATH="$PATH:/home/alice"
prompt$ hello
-bash: hello: Permission denied
prompt$
```

The "Hello, world" of shell scripts

- Let's build a simple script and see how to make it run
- Is this enough? No. What happened?
- We need to have execute permission for `hello`

```
prompt$ hello
-bash: hello: command not found
prompt$ pwd
/home/alice
prompt$ PATH="$PATH:/home/alice"
prompt$ hello
-bash: hello: Permission denied
prompt$
```

The "Hello, world" of shell scripts

- Let's build a simple script and see how to make it run

```
prompt$ hello
-bash: hello: command not found
prompt$ pwd
/home/alice
prompt$ PATH="$PATH:/home/alice"
prompt$ hello
-bash: hello: Permission denied
prompt$ chmod +x hello
```

The "Hello, world" of shell scripts

- Let's build a simple script and see how to make it run
- Is this enough? Let's try.

```
-bash: hello: command not found
prompt$ pwd
/home/alice
prompt$ PATH="$PATH:/home/alice"
prompt$ hello
-bash: hello: Permission denied
prompt$ chmod +x hello
prompt$
```

The "Hello, world" of shell scripts

- Let's build a simple script and see how to make it run
- Is this enough? Let's try.

```
-bash: hello: command not found
prompt$ pwd
/home/alice
prompt$ PATH="$PATH:/home/alice"
prompt$ hello
-bash: hello: Permission denied
prompt$ chmod +x hello
prompt$ hello
```

The "Hello, world" of shell scripts

- Let's build a simple script and see how to make it run
- Is this enough? Usually. But there is an issue...

```
/home/alice
prompt$ PATH="$PATH:/home/alice"
prompt$ hello
-bash: hello: Permission denied
prompt$ chmod +x hello
prompt$ hello
Hello, world!
prompt$
```

What's wrong with our `hello` script?

- Let's assume the script will execute when invoked
- `hello` has commands to run in bash

What's wrong with our `hello` script?

- Let's assume the script will execute when invoked
- `hello` has commands to run in bash
- What if we are running a different shell?
  - Any "`bash` specific" statements will cause our shell to barf

What's wrong with our `hello` script?

- Let's assume the script will execute when invoked
- `hello` has commands to run in bash
- What if we are running a different shell?
  - Any "`bash` specific" statements will cause our shell to barf
- Ok, so how do we guarantee our script is run in bash?

Some UNIX magic

- If I have a text file with execute permission turned on
- AND the first two characters are "#!"
- THEN, when I type the name of the file:

Some UNIX magic

- If I have a text file with execute permission turned on
- AND the first two characters are "#!"
- THEN, when I type the name of the file:

1. The rest of the first line is interpreted as a utility to run
2. The utility is run with the name of the file appended

Example (1): UNIX magic

```
prompt$
```

Example (1): UNIX magic

```
prompt$ cat > foo.txt
```

Example (1): UNIX magic

```
prompt$ cat > foo.txt
```

Example (1): UNIX magic

```
prompt$ cat > foo.txt
#!/bin/cat -n
```

Example (1): UNIX magic

```
prompt$ cat > foo.txt
#!/bin/cat -n
```

Example (1): UNIX magic

```
prompt$ cat > foo.txt
#!/bin/cat -n
My bizarre file.
```

Example (1): UNIX magic

```
prompt$ cat > foo.txt
#!/bin/cat -n
My bizarre file.
```

Example (1): UNIX magic

```
prompt$ cat > foo.txt
#!/bin/cat -n
My bizarre file.
Such as it is.
```

Example (1): UNIX magic

```
prompt$ cat > foo.txt
#!/bin/cat -n
My bizarre file.
Such as it is.
```

Example (1): UNIX magic

```
prompt$ cat > foo.txt
#!/bin/cat -n
My bizarre file.
Such as it is.
prompt$
```

Example (1): UNIX magic

```
prompt$ cat > foo.txt
#!/bin/cat -n
My bizarre file.
Such as it is.
prompt$ chmod +x foo.txt
```

Example (1): UNIX magic

```
prompt$ cat > foo.txt
#!/bin/cat -n
My bizarre file.
Such as it is.
prompt$ chmod +x foo.txt
prompt$
```

Example (1): UNIX magic

```
prompt$ cat > foo.txt
#!/bin/cat -n
My bizarre file.
Such as it is.
prompt$ chmod +x foo.txt
prompt$ ./foo.txt
```

Example (1): UNIX magic

```
prompt$ cat > foo.txt
#!/bin/cat -n
My bizarre file.
Such as it is.
prompt$ chmod +x foo.txt
prompt$ ./foo.txt
   1  #!/bin/cat -n
   2  My bizarre file.
   3  Such as it is.
prompt$
```

Example (2): UNIX magic

```
prompt$
```

Example (2): UNIX magic

```
prompt$ cat > fu.txt
```

Example (2): UNIX magic

```
prompt$ cat > fu.txt
```

Example (2): UNIX magic

```
prompt$ cat > fu.txt
#!/bin/echo This example is
```

Example (2): UNIX magic

```
prompt$ cat > fu.txt
#!/bin/echo This example is
```

Example (2): UNIX magic

```
prompt$ cat > fu.txt
#!/bin/echo This example is
An even more bizarre file.
```

Example (2): UNIX magic

```
prompt$ cat > fu.txt
#!/bin/echo This example is
An even more bizarre file.
```

Example (2): UNIX magic

```
prompt$ cat > fu.txt
#!/bin/echo This example is
An even more bizarre file.
prompt$
```

Example (2): UNIX magic

```
prompt$ cat > fu.txt
#!/bin/echo This example is
An even more bizarre file.
prompt$ chmod +x fu.txt
```

Example (2): UNIX magic

```
prompt$ cat > fu.txt
#!/bin/echo This example is
An even more bizarre file.
prompt$ chmod +x fu.txt
prompt$
```

Example (2): UNIX magic

```
prompt$ cat > fu.txt
#!/bin/echo This example is
An even more bizarre file.
prompt$ chmod +x fu.txt
prompt$ ./fu.txt
```

Example (2): UNIX magic

```
prompt$ cat > fu.txt
#!/bin/echo This example is
An even more bizarre file.
prompt$ chmod +x fu.txt
prompt$ ./fu.txt
This example is ./fu.txt
prompt$
```

Example (3): UNIX magic

```
prompt$
```

Example (3): UNIX magic

```
prompt$ cat > bar
```

Example (3): UNIX magic

```
prompt$ cat > bar
```

Example (3): UNIX magic

```
prompt$ cat > bar
#!/bin/ls -l
```

Example (3): UNIX magic

```
prompt$ cat > bar
#!/bin/ls -l
```

Example (3): UNIX magic

```
prompt$ cat > bar
#!/bin/ls -l
Last crazy one, I promise.
```

Example (3): UNIX magic

```
prompt$ cat > bar
#!/bin/ls -l
Last crazy one, I promise.
```

Example (3): UNIX magic

```
prompt$ cat > bar
#!/bin/ls -l
Last crazy one, I promise.
prompt$
```

Example (3): UNIX magic

```
prompt$ cat > bar
#!/bin/ls -l
Last crazy one, I promise.
prompt$ chmod +x bar
```

Example (3): UNIX magic

```
prompt$ cat > bar
#!/bin/ls -l
Last crazy one, I promise.
prompt$ chmod +x bar
prompt$
```

Example (3): UNIX magic

```
prompt$ cat > bar
#!/bin/ls -l
Last crazy one, I promise.
prompt$ chmod +x bar
prompt$ ./bar
```

Example (3): UNIX magic

```
prompt$ cat > bar
#!/bin/ls -l
Last crazy one, I promise.
prompt$ chmod +x bar
prompt$ ./bar
-rwx------  1 alice  staff  40 Sep 14 17:07 ./bar
prompt$
```

Shell scripts

1. How do I write a script that always runs under `bash`?

Shell scripts

1. How do I write a script that always runs under `bash`?
   - Easy — use the following for the first line:

```
#!/bin/bash
```

Shell scripts

1. How do I write a script that always runs under `bash`?
   - Easy — use the following for the first line:

```
#!/bin/bash
```

2. But what will `bash` do when it executes the first line?

Shell scripts

1. How do I write a script that always runs under `bash`?
   - Easy — use the following for the first line:

```
#!/bin/bash
```

2. But what will `bash` do when it executes the first line?
   - Any line beginning with "#" is ignored by the shell
   - So the first line will be ignored

The final, proper, hello world script

```
prompt$
```

The final, proper, hello world script

```
prompt$ cat > hello
```

The final, proper, hello world script

```
prompt$ cat > hello
```

The final, proper, hello world script

```
prompt$ cat > hello
#!/bin/bash
```

The final, proper, hello world script

```
prompt$ cat > hello
#!/bin/bash
```

The final, proper, hello world script

```
prompt$ cat > hello
#!/bin/bash

```

The final, proper, hello world script

```
prompt$ cat > hello
#!/bin/bash

echo 'Hello, world!'
```

The final, proper, hello world script

```
prompt$ cat > hello
#!/bin/bash

echo 'Hello, world!'
```

The final, proper, hello world script

```
prompt$ cat > hello
#!/bin/bash

echo 'Hello, world!'
prompt$
```

The final, proper, hello world script

```
prompt$ cat > hello
#!/bin/bash

echo 'Hello, world!'
prompt$ chmod +x hello
```

The final, proper, hello world script

```
prompt$ cat > hello
#!/bin/bash

echo 'Hello, world!'
prompt$ chmod +x hello
prompt$
```

The final, proper, hello world script

```
prompt$ cat > hello
#!/bin/bash

echo 'Hello, world!'
prompt$ chmod +x hello
prompt$ ./hello
```

The final, proper, hello world script

```
prompt$ cat > hello
#!/bin/bash

echo 'Hello, world!'
prompt$ chmod +x hello
prompt$ ./hello
Hello, world!
prompt$
```

A slightly fancier hello script

Want: Prompt for user's name (to `stderr`), and say hello.

```
prompt$
```

A slightly fancier hello script

Want: Prompt for user's name (to `stderr`), and say hello.

```
prompt$ cat > hello2
```

A slightly fancier hello script

Want: Prompt for user's name (to `stderr`), and say hello.

```
prompt$ cat > hello2
```

A slightly fancier hello script

Want: Prompt for user's name (to `stderr`), and say hello.

```
prompt$ cat > hello2
#!/bin/bash
```

A slightly fancier hello script

Want: Prompt for user's name (to `stderr`), and say hello.

```
prompt$ cat > hello2
#!/bin/bash
```

A slightly fancier hello script

Want: Prompt for user's name (to `stderr`), and say hello.

```
prompt$ cat > hello2
#!/bin/bash

```

A slightly fancier hello script

Want: Prompt for user's name (to `stderr`), and say hello.

```
prompt$ cat > hello2
#!/bin/bash

echo 'What is your name?'
```

A slightly fancier hello script

Want: Prompt for user's name (to `stderr`), and say hello.

```
prompt$ cat > hello2
#!/bin/bash

echo 'What is your name?' 1>&2
```

A slightly fancier hello script

Want: Prompt for user's name (to `stderr`), and say hello.

```
prompt$ cat > hello2
#!/bin/bash

echo 'What is your name?' 1>&2
```

A slightly fancier hello script

Want: Prompt for user's name (to `stderr`), and say hello.

```
prompt$ cat > hello2
#!/bin/bash

echo 'What is your name?' 1>&2
read name
```

A slightly fancier hello script

Want: Prompt for user's name (to `stderr`), and say hello.

```
prompt$ cat > hello2
#!/bin/bash

echo 'What is your name?' 1>&2
read name
```

A slightly fancier hello script

Want: Prompt for user's name (to `stderr`), and say hello.

```
prompt$ cat > hello2
#!/bin/bash

echo 'What is your name?' 1>&2
read name
echo "Hello, $name."
```

A slightly fancier hello script

Want: Prompt for user's name (to `stderr`), and say hello.

```
prompt$ cat > hello2
#!/bin/bash

echo 'What is your name?' 1>&2
read name
echo "Hello, $name."
```

A slightly fancier hello script

Want: Prompt for user's name (to `stderr`), and say hello.

```
prompt$ cat > hello2
#!/bin/bash

echo 'What is your name?' 1>&2
read name
echo "Hello, $name."
prompt$
```

A slightly fancier hello script

Want: Prompt for user's name (to `stderr`), and say hello.

```
prompt$ cat > hello2
#!/bin/bash

echo 'What is your name?' 1>&2
read name
echo "Hello, $name."
prompt$ chmod +x hello2
```

A slightly fancier hello script

Want: Prompt for user's name (to `stderr`), and say hello.

```
prompt$ cat > hello2
#!/bin/bash

echo 'What is your name?' 1>&2
read name
echo "Hello, $name."
prompt$ chmod +x hello2
prompt$
```

A slightly fancier hello script

Want: Prompt for user's name (to `stderr`), and say hello.

```
prompt$ cat > hello2
#!/bin/bash

echo 'What is your name?' 1>&2
read name
echo "Hello, $name."
prompt$ chmod +x hello2
prompt$ ./hello2
```

A slightly fancier hello script

Want: Prompt for user's name (to `stderr`), and say hello.

```
prompt$ cat > hello2
#!/bin/bash

echo 'What is your name?' 1>&2
read name
echo "Hello, $name."
prompt$ chmod +x hello2
prompt$ ./hello2
What is your name?
```

A slightly fancier hello script

Want: Prompt for user's name (to `stderr`), and say hello.

```
prompt$ cat > hello2
#!/bin/bash

echo 'What is your name?' 1>&2
read name
echo "Hello, $name."
prompt$ chmod +x hello2
prompt$ ./hello2
What is your name?
Bob
```

A slightly fancier hello script

Want: Prompt for user's name (to `stderr`), and say hello.

```
prompt$ cat > hello2
#!/bin/bash

echo 'What is your name?' 1>&2
read name
echo "Hello, $name."
prompt$ chmod +x hello2
prompt$ ./hello2
What is your name?
Bob
Hello, Bob.
prompt$
```

Debugging shell scripts

- Debugging shell scripts is tricky
  - We won't write long ones
- Helpful hint: if you run a script with `bash -x script`

  It displays what it will execute, before executing it

```
prompt$
```

Debugging shell scripts

- Debugging shell scripts is tricky
  - We won't write long ones
- Helpful hint: if you run a script with `bash -x script`

  It displays what it will execute, before executing it

```
prompt$ bash -x hello2
```

Debugging shell scripts

- Debugging shell scripts is tricky
  - We won't write long ones
- Helpful hint: if you run a script with `bash -x script`

  It displays what it will execute, before executing it

```
prompt$ bash -x hello2
+ echo 'What is your name?'
What is your name?
+ read name
```

Debugging shell scripts

- Debugging shell scripts is tricky
  - We won't write long ones
- Helpful hint: if you run a script with `bash -x script`

  It displays what it will execute, before executing it

```
prompt$ bash -x hello2
+ echo 'What is your name?'
What is your name?
+ read name
Bob
```

Debugging shell scripts

- Debugging shell scripts is tricky
  - We won't write long ones
- Helpful hint: if you run a script with `bash -x script`

  It displays what it will execute, before executing it

```
prompt$ bash -x hello2
+ echo 'What is your name?'
What is your name?
+ read name
Bob
+ echo 'Hello, Bob'
Hello, Bob
prompt$
```

Revisiting the `PATH` issue

Why did I keep typing `./hello` instead of simply `hello`?

- Because the directory containing `hello` is not in the `PATH`

Revisiting the `PATH` issue

Why did I keep typing `./hello` instead of simply `hello`?

- Because the directory containing `hello` is not in the `PATH`

That's annoying. How can I keep from typing `./` all the time?

Revisiting the PATH issue

Why did I keep typing `./hello` instead of simply `hello`?

- Because the directory containing `hello` is not in the `PATH`

That's annoying. How can I keep from typing `./` all the time?

1. Add *all* my directories to `PATH`
   - Not practical
   - Huge `PATH` means slow shell (why?)

Revisiting the PATH issue

Why did I keep typing `./hello` instead of simply `hello`?

- Because the directory containing `hello` is not in the `PATH`

That's annoying. How can I keep from typing `./` all the time?

1. Add *all* my directories to `PATH`
   - Not practical
   - Huge `PATH` means slow shell (why?)
2. Add `.` to `PATH`
   - This works and will not be slow, but...
   - NOT SAFE

Revisiting the PATH issue

Why did I keep typing `./hello` instead of simply `hello`?

- Because the directory containing `hello` is not in the `PATH`

That's annoying. How can I keep from typing `./` all the time?

1. Add *all* my directories to `PATH`
   - Not practical
   - Huge `PATH` means slow shell (why?)
2. Add `.` to `PATH`
   - This works and will not be slow, but...
   - NOT SAFE
3. Put frequently–used executables in one directory
   - E.g., in `~/bin`; include this in your `PATH`
   - You can put symlinks to the actual executables

Example: why . in your PATH is unsafe

- Suppose `chuck` is evil but nobody knows
- `chuck` says something like:

  "I put a bunch of cool things in `~chuck/cool`, check it out."
- Suppose `bob` has `PATH=.:/usr/local/bin:/bin:/usr/bin`
- Suppose `bob` does:

```
prompt$
```

Example: why . in your PATH is unsafe

- Suppose chuck is evil but nobody knows
- chuck says something like:

  "I put a bunch of cool things in ~chuck/cool, check it out."
- Suppose bob has PATH=.:/usr/local/bin:/bin:/usr/bin
- Suppose bob does:

```
prompt$ cd ~chuck/cool
```

Example: why . in your PATH is unsafe

- Suppose chuck is evil but nobody knows
- chuck says something like:

  "I put a bunch of cool things in ~chuck/cool, check it out."
- Suppose bob has PATH=.:/usr/local/bin:/bin:/usr/bin
- Suppose bob does:

```
prompt$ cd ~chuck/cool
prompt$
```

Example: why . in your PATH is unsafe

- Suppose chuck is evil but nobody knows
- chuck says something like:

  "I put a bunch of cool things in ~chuck/cool, check it out."
- Suppose bob has PATH=.:/usr/local/bin:/bin:/usr/bin
- Suppose bob does:

```
prompt$ cd ~chuck/cool
prompt$ ls
```

Example: why . in your PATH is unsafe

- Suppose chuck is evil but nobody knows
- chuck says something like:

  "I put a bunch of cool things in ~chuck/cool, check it out."
- Suppose bob has PATH=.:/usr/local/bin:/bin:/usr/bin
- Suppose bob does:

```
prompt$ cd ~chuck/cool
prompt$ ls
bofh01.txt   bofh04.txt   coolgame02
bofh02.txt   bofh05.txt   neatofile
bofh03.txt   coolgame01   supercool1
prompt$
```

- chuck now owns bob's account
- bob most likely does not notice

Huh? Why does `chuck` own `bob` now?

Because the coolest thing in `~chuck/cool` was not displayed:

Huh? Why does `chuck` own `bob` now?

Because the coolest thing in `~chuck/cool` was not displayed:

- `chuck` has a shell script named `ls`
- It was `chuck's` `ls` that executed, not the "real" one
- What did the diabolic `ls` do?

Huh? Why does `chuck` own `bob` now?

Because the coolest thing in `~chuck/cool` was not displayed:

- `chuck` has a shell script named `ls`
- It was `chuck`'s `ls` that executed, not the "real" one
- What did the diabolic `ls` do?
  1. Copy `chuck`'s ssh key into `bob`'s home directory
     - `chuck` can now `ssh` into `bob`'s account, whenever
     - Without typing `bob`'s password

Huh? Why does `chuck` own `bob` now?

Because the coolest thing in `~chuck/cool` was not displayed:

- `chuck` has a shell script named `ls`
- It was `chuck`'s `ls` that executed, not the "real" one
- What did the diabolic `ls` do?
  1. Copy `chuck`'s ssh key into `bob`'s home directory
     - `chuck` can now `ssh` into `bob`'s account, whenever
     - Without typing `bob`'s password
  2. Run the "real" `ls`

Huh? Why does `chuck` own `bob` now?

Because the coolest thing in `~chuck/cool` was not displayed:

- `chuck` has a shell script named `ls`
- It was `chuck`'s `ls` that executed, not the "real" one
- What did the diabolic `ls` do?
  1. Copy `chuck`'s ssh key into `bob`'s home directory
     - `chuck` can now `ssh` into `bob`'s account, whenever
     - Without typing `bob`'s password
  2. Run the "real" `ls`
     - But only show the files that `chuck` wants to be seen

Huh? Why does `chuck` own `bob` now?

Because the coolest thing in `~chuck/cool` was not displayed:

- `chuck` has a shell script named `ls`
- It was `chuck`'s `ls` that executed, not the "real" one
- What did the diabolic `ls` do?
  1. Copy `chuck`'s ssh key into `bob`'s home directory
     - `chuck` can now `ssh` into `bob`'s account, whenever
     - Without typing `bob`'s password
  2. Run the "real" `ls`
     - But only show the files that `chuck` wants to be seen
     - And if `chuck` is *really* clever...
     - Determines `bob`'s alias for `ls` so the output is right

Huh? Why does `chuck` own `bob` now?

Because the coolest thing in `~chuck/cool` was not displayed:

- `chuck` has a shell script named `ls`
- It was `chuck`'s `ls` that executed, not the "real" one
- What did the diabolic `ls` do?
  1. Copy `chuck`'s ssh key into `bob`'s home directory
     - `chuck` can now `ssh` into `bob`'s account, whenever
     - Without typing `bob`'s password
  2. Run the "real" `ls`
     - But only show the files that `chuck` wants to be seen
     - And if `chuck` is *really* clever...
     - Determines `bob`'s alias for `ls` so the output is right
- `chuck`'s evil laughter here

Wait, can `chuck` copy files into `bob`'s home?

- Let's look:

```
prompt$
```

Wait, can `chuck` copy files into `bob`'s home?

- Let's look:

```
prompt$ ls -ld ~bob
```

Wait, can chuck copy files into bob's home?

- Let's look:

```
prompt$ ls -ld ~bob
drwx------ 50 bob  staff 3896 Sep 15 23:07 /home/bob/
prompt$
```

- No, chuck cannot copy files into bob's home.
  - If he could, no need to craft the diabolic ls script

Wait, can chuck copy files into bob's home?

- Let's look:

```
prompt$ ls -ld ~bob
drwx------ 50 bob  staff 3896 Sep 15 23:07 /home/bob/
prompt$
```

- No, chuck cannot copy files into bob's home.
  - If he could, no need to craft the diabolic ls script
- Ok, then how does diabolic ls copy chuck's ssh key?

Wait, can chuck copy files into bob's home?

- Let's look:

```
prompt$ ls -ld ~bob
drwx------ 50 bob  staff 3896 Sep 15 23:07 /home/bob/
prompt$
```

- No, chuck cannot copy files into bob's home.
  - If he could, no need to craft the diabolic ls script
- Ok, then how does diabolic ls copy chuck's ssh key?
- Easy — bob ran that script, not chuck
  - Everything in that script runs as bob
  - bob is running a script that chuck wrote
  - That's why chuck wanted to trick bob into executing it

How to tell what runs

`which`: search `PATH` for names

- Usage: `which [cmd] [cmd] ...`
- For each command, show the full pathname of what would be executed

Back to `bob` and the diabolic `ls`:

```
prompt$
```

How to tell what runs

`which`: search `PATH` for names

- Usage: `which [cmd] [cmd] ...`
- For each command, show the full pathname of what would be executed

Back to `bob` and the diabolic `ls`:

```
prompt$ cd
```

How to tell what runs

`which`: search `PATH` for names

- Usage: `which [cmd] [cmd] ...`
- For each command, show the full pathname of what would be executed

Back to `bob` and the diabolic `ls`:

```
prompt$ cd
prompt$
```

How to tell what runs

`which`: search `PATH` for names

- Usage: `which [cmd] [cmd] ...`
- For each command, show the full pathname of what would be executed

Back to `bob` and the diabolic `ls`:

```
prompt$ cd
prompt$ echo $PATH
```

How to tell what runs

`which`: search `PATH` for names

- Usage: `which [cmd] [cmd] ...`
- For each command, show the full pathname of what would be executed

Back to `bob` and the diabolic `ls`:

```
prompt$ cd
prompt$ echo $PATH
.:/usr/local/bin:/bin/:/usr/bin
prompt$
```

How to tell what runs

`which`: search `PATH` for names

- Usage: `which [cmd] [cmd] ...`
- For each command, show the full pathname of what would be executed

Back to `bob` and the diabolic `ls`:

```
prompt$ cd
prompt$ echo $PATH
.:/usr/local/bin:/bin/:/usr/bin
prompt$ which ls
```

How to tell what runs

`which`: search `PATH` for names

- Usage: `which [cmd] [cmd] ...`
- For each command, show the full pathname of what would be executed

Back to `bob` and the diabolic `ls`:

```
prompt$ cd
prompt$ echo $PATH
.:/usr/local/bin:/bin/:/usr/bin
prompt$ which ls
/bin/ls
prompt$
```

How to tell what runs

`which`: search `PATH` for names

- Usage: `which [cmd] [cmd] ...`
- For each command, show the full pathname of what would be executed

Back to `bob` and the diabolic `ls`:

```
prompt$ cd
prompt$ echo $PATH
.:/usr/local/bin:/bin/:/usr/bin
prompt$ which ls
/bin/ls
prompt$ cd ~chuck/cool
```

How to tell what runs

`which`: search `PATH` for names

- Usage: `which [cmd] [cmd] ...`
- For each command, show the full pathname of what would be executed

Back to `bob` and the diabolic `ls`:

```
prompt$ echo $PATH
.:/usr/local/bin:/bin/:/usr/bin
prompt$ which ls
/bin/ls
prompt$ cd ~chuck/cool
prompt$
```

How to tell what runs

`which`: search `PATH` for names

- Usage: `which [cmd] [cmd] ...`
- For each command, show the full pathname of what would be executed

Back to `bob` and the diabolic `ls`:

```
prompt$ echo $PATH
.:/usr/local/bin:/bin/:/usr/bin
prompt$ which ls
/bin/ls
prompt$ cd ~chuck/cool
prompt$ which ls
```

How to tell what runs

`which`: search `PATH` for names

- Usage: `which [cmd] [cmd] ...`
- For each command, show the full pathname of what would be executed

Back to `bob` and the diabolic `ls`:

```
prompt$ which ls
/bin/ls
prompt$ cd ~chuck/cool
prompt$ which ls
./ls
prompt$
```

How to tell what runs

`which`: search `PATH` for names

- Usage: `which [cmd] [cmd] ...`
- For each command, show the full pathname of what would be executed

Back to `bob` and the diabolic `ls`:

```
prompt$ which ls
/bin/ls
prompt$ cd ~chuck/cool
prompt$ which ls
./ls
prompt$ /bin/ls
```

How to tell what runs

`which`: search `PATH` for names

- Usage: `which [cmd] [cmd] ...`
- For each command, show the full pathname of what would be executed

Back to `bob` and the diabolic `ls`:

```
./ls
prompt$ /bin/ls
bofh01.txt  bofh04.txt  coolgame02  supercool1
bofh02.txt  bofh05.txt  ls
bofh03.txt  coolgame01  neatofile
prompt$
```

What if . is at the *end* of the PATH?

- Harder for `chuck` but still possible
- Now, `chuck` has to guess a utility that `bob` will
  - mistype (e.g., `sl` or `la` instead of `ls`); or
  - type correctly but is not available
- This script should
  1. Copy the ssh key
  2. Print "`bash: la: command not found`"
     - If `chuck` is clever, print the proper message for various shells

What if . is at the *end* of the PATH?

- Harder for `chuck` but still possible
- Now, `chuck` has to guess a utility that `bob` will
  - mistype (e.g., `sl` or `la` instead of `ls`); or
  - type correctly but is not available
- This script should
  1. Copy the ssh key
  2. Print "`bash: la: command not found`"
     - If `chuck` is clever, print the proper message for various shells
- Actually, for this example, it is easier for `chuck` to:
  - Modify `coolgame01` and `coolgame02` to copy the ssh key before running the game
  - This is called a "Trojan Horse"

Paranoid yet?

- Yes? Good.
- And that's why we NEVER PUT . IN OUR `PATH`

Shell script arguments

- Suppose we want a script that can read its arguments, e.g.:

```
prompt$
```

- Need a way to get the arguments
  - And the logic to process them
    (We will discuss that later)
- There are a few ways to deal with arguments
- We will start with a simple one

Shell script arguments

- Suppose we want a script that can read its arguments, e.g.:

```
prompt$ ./hello Bob
```

- Need a way to get the arguments
  - And the logic to process them
    (We will discuss that later)
- There are a few ways to deal with arguments
- We will start with a simple one

Shell script arguments

- Suppose we want a script that can read its arguments, e.g.:

```
prompt$ ./hello Bob
Hello, Bob!
prompt$
```

- Need a way to get the arguments
  - And the logic to process them
    (We will discuss that later)
- There are a few ways to deal with arguments
- We will start with a simple one

Shell script arguments

- Suppose we want a script that can read its arguments, e.g.:

```
prompt$ ./hello Bob
Hello, Bob!
prompt$ ./hello -i Roberto
```

- Need a way to get the arguments
  - And the logic to process them
    (We will discuss that later)
- There are a few ways to deal with arguments
- We will start with a simple one

Shell script arguments

- Suppose we want a script that can read its arguments, e.g.:

```
prompt$ ./hello Bob
Hello, Bob!
prompt$ ./hello -i Roberto
Ciao, Roberto!
prompt$
```

- Need a way to get the arguments
  - And the logic to process them
    (We will discuss that later)
- There are a few ways to deal with arguments
- We will start with a simple one

Special shell variables

`$1` : the first argument passed

`$2` : the second argument passed

⋮

`$9` : the ninth argument passed

Special shell variables

`$1` : the first argument passed

`$2` : the second argument passed

⋮

`$9` : the ninth argument passed

`$#` : the number of arguments passed

`$@` : the entire argument string
(first argument and later)

Special shell variables

`$1` : the first argument passed

`$2` : the second argument passed

⋮

`$9` : the ninth argument passed

`$#` : the number of arguments passed

`$@` : the entire argument string (first argument and later)

`$0` : the zeroth argument (?)

Special shell variables

`$1` : the first argument passed

`$2` : the second argument passed

⋮

`$9` : the ninth argument passed

`$#` : the number of arguments passed

`$@` : the entire argument string (first argument and later)

`$0` : the zeroth argument (?)

Let's see how these work . . .

Argument test

Argument test

Shell script: args

```
#!/bin/bash
echo "zeroth: $0"; echo "first : $1"
echo "second: $2"; echo "third : $3"
echo "number: $#"; echo "all   : $@"
```

Argument test

Shell script: args

```
#!/bin/bash
echo "zeroth: $0"; echo "first : $1"
echo "second: $2"; echo "third : $3"
echo "number: $#"; echo "all   : $@"
```

```
prompt$
```

Argument test

Shell script: args

```
#!/bin/bash
echo "zeroth: $0"; echo "first : $1"
echo "second: $2"; echo "third : $3"
echo "number: $#"; echo "all   : $@"
```

```
prompt$ pwd
```

Argument test

Shell script: args

```
#!/bin/bash
echo "zeroth: $0"; echo "first : $1"
echo "second: $2"; echo "third : $3"
echo "number: $#"; echo "all   : $@"
```

```
prompt$ pwd
/home/alice/scripts
prompt$
```

Argument test

Shell script: args

```
#!/bin/bash
echo "zeroth: $0"; echo "first : $1"
echo "second: $2"; echo "third : $3"
echo "number: $#"; echo "all   : $@"
```

```
prompt$ pwd
/home/alice/scripts
prompt$ ~/scripts/args foo bar
```

Argument test

Shell script: args

```
#!/bin/bash
echo "zeroth: $0"; echo "first : $1"
echo "second: $2"; echo "third : $3"
echo "number: $#"; echo "all   : $@"
```

```
prompt$ ~/scripts/args foo bar
zeroth: /home/alice/scripts/args
first : foo
second: bar
third :
number: 2
all   : foo bar
prompt$
```

Argument test

Shell script: args

```
#!/bin/bash
echo "zeroth: $0"; echo "first : $1"
echo "second: $2"; echo "third : $3"
echo "number: $#"; echo "all   : $@"
```

```
prompt$ ~/scripts/args foo bar
zeroth: /home/alice/scripts/args
first : foo
second: bar
third :
number: 2
all   : foo bar
prompt$ ./args 'What happens if' we 'group things?'
```

Argument test

Shell script: args

```
#!/bin/bash
echo "zeroth: $0"; echo "first : $1"
echo "second: $2"; echo "third : $3"
echo "number: $#"; echo "all   : $@"
```

```
prompt$ ./args 'What happens if' we 'group things?'
zeroth: ./args
first : What happens if
second: we
third : group things?
number: 3
all   : What happens if we group things?
prompt$
```

Argument test

Shell script: args

```
#!/bin/bash
echo "zeroth: $0"; echo "first : $1"
echo "second: $2"; echo "third : $3"
echo "number: $#"; echo "all   : $@"
```

```
prompt$ ./args 'What happens if' we 'group things?'
zeroth: ./args
first : What happens if
second: we
third : group things?
number: 3
all   : What happens if we group things?
prompt$ ./args a b c d e f g h i j k l m n o p
```

Argument test

Shell script: args

```
#!/bin/bash
echo "zeroth: $0"; echo "first : $1"
echo "second: $2"; echo "third : $3"
echo "number: $#"; echo "all   : $@"
```

```
prompt$ ./args a b c d e f g h i j k l m n o p
zeroth: ./args
first : a
second: b
third : c
number: 16
all   : a b c d e f g h i j k l m n o p
prompt$
```

Argument test

Shell script: args

```
#!/bin/bash
echo "zeroth: $0"; echo "first : $1"
echo "second: $2"; echo "third : $3"
echo "number: $#"; echo "all   : $@"
```

```
prompt$ ./args a b c d e f g h i j k l m n o p
zeroth: ./args
first : a
second: b
third : c
number: 16
all   : a b c d e f g h i j k l m n o p
prompt$ ./args *
```

Argument test

Shell script: args

```
#!/bin/bash
echo "zeroth: $0"; echo "first : $1"
echo "second: $2"; echo "third : $3"
echo "number: $#"; echo "all   : $@"
```

```
prompt$ ./args *
zeroth: ./args
first : args
second: hello
third : hello2
number: 4
all   : args hello hello2 Readme
prompt$
```

Questions about arguments

- What if there are more than 9 arguments?
  - Use braces, e.g., `${15}`
  - But with this many arguments, you should probably be doing something else
- So $15 won't get the 15th argument?
  - No, it gives you the first, with a 5 added.
- Is there a good way to process optional switches?
  - `getopts` works pretty well
  - . . . but we will discuss that later, also
- Will the arguments move if I invoke the script differently? E.g., `bash -x myscript args`
  - No change — `bash` is clever that way

Can I call one shell script from another?

Can I call one shell script from another?

- Yes, exactly as you would expect

Can I call one shell script from another?

- Yes, exactly as you would expect
- But there is a path problem . . .

Example file: /home/alice/scripts/count

```
#!/bin/bash
echo $#
```

File: /home/alice/scripts/name

```
#!/bin/bash
read -p "What is your name?!" name
echo "Your name has ‘./count $name‘ words."
```

```
prompt$
```

Example file: /home/alice/scripts/count

```
#!/bin/bash
echo $#
```

File: /home/alice/scripts/name

```
#!/bin/bash
read -p "What is your name?!" name
echo "Your name has ‘./count $name‘ words."
```

```
prompt$ ./name
```

Example file: /home/alice/scripts/count

```
#!/bin/bash
echo $#
```

File: /home/alice/scripts/name

```
#!/bin/bash
read -p "What is your name?!" name
echo "Your name has ‘./count $name‘ words."
```

```
prompt$ ./name
What is your name?
```

Example file: /home/alice/scripts/count

```
#!/bin/bash
echo $#
```

File: /home/alice/scripts/name

```
#!/bin/bash
read -p "What is your name?!" name
echo "Your name has ‘./count $name‘ words."
```

```
prompt$ ./name
What is your name? The artist formerly known as prince
```

Example file: /home/alice/scripts/count

```
#!/bin/bash
echo $#
```

File: /home/alice/scripts/name

```
#!/bin/bash
read -p "What is your name?!" name
echo "Your name has ‘./count $name‘ words."
```

```
prompt$ ./name
What is your name? The artist formerly known as prince
Your name has 6 words.
prompt$
```

Example file: /home/alice/scripts/count

```
#!/bin/bash
echo $#
```

File: /home/alice/scripts/name

```
#!/bin/bash
read -p "What is your name?!" name
echo "Your name has ‘./count $name‘ words."
```

```
prompt$ ./name
What is your name? The artist formerly known as prince
Your name has 6 words.
prompt$ cd ..
```

Example file: /home/alice/scripts/count

```
#!/bin/bash
echo $#
```

File: /home/alice/scripts/name

```
#!/bin/bash
read -p "What is your name?!" name
echo "Your name has ‘./count $name‘ words."
```

```
prompt$ ./name
What is your name? The artist formerly known as prince
Your name has 6 words.
prompt$ cd ..
prompt$
```

Example file: /home/alice/scripts/count

```
#!/bin/bash
echo $#
```

File: /home/alice/scripts/name

```
#!/bin/bash
read -p "What is your name?!" name
echo "Your name has ‘./count $name‘ words."
```

```
prompt$ ./name
What is your name? The artist formerly known as prince
Your name has 6 words.
prompt$ cd ..
prompt$ scripts/name
```

Example file: /home/alice/scripts/count

```
#!/bin/bash
echo $#
```

File: /home/alice/scripts/name

```
#!/bin/bash
read -p "What is your name?!" name
echo "Your name has ‘./count $name‘ words."
```

```
What is your name? The artist formerly known as prince
Your name has 6 words.
prompt$ cd ..
prompt$ scripts/name
What is your name?
```

Example file: /home/alice/scripts/count

```
#!/bin/bash
echo $#
```

File: /home/alice/scripts/name

```
#!/bin/bash
read -p "What is your name?!" name
echo "Your name has ‘./count $name‘ words."
```

```
What is your name? The artist formerly known as prince
Your name has 6 words.
prompt$ cd ..
prompt$ scripts/name
What is your name? Doctor Bob Roberts
```

Example file: /home/alice/scripts/count

```
#!/bin/bash
echo $#
```

File: /home/alice/scripts/name

```
#!/bin/bash
read -p "What is your name?!" name
echo "Your name has ‘./count $name‘ words."
```

```
prompt$ scripts/name
What is your name? Doctor Bob Roberts
scripts/name: line 3: ./count: No such file or directory
Your name has  words.
prompt$
```

Fixing the path problem

1. Use absolute paths when calling other scripts
   - Super annoying
   - Painful if you move scripts

Fixing the path problem

1. Use absolute paths when calling other scripts
   - Super annoying
   - Painful if you move scripts
2. Ensure critical scripts are in your `PATH`
   - Limits where your scripts can live
     because you shouldn't add `.` to your `PATH`

Fixing the path problem

1. Use absolute paths when calling other scripts
   - Super annoying
   - Painful if you move scripts
2. Ensure critical scripts are in your `PATH`
   - Limits where your scripts can live
     because you shouldn't add `.` to your `PATH`
3. Use shell *functions*
   - Most of the time, this is what you meant to do anyway

Functions in bash

Defining a function:

```
function funcname() # ‘‘function’’ is optional
{
    Normal shell commands here
    Perhaps several lines of them
}
```

Calling a function: just like calling another script

- Pretend the function name is the name of the file
- Pretend the script is in your path

Function parameters:

- Just like writing your function in another script
- DO NOT need to change your function header

Converting the last example into a function

File: /home/alice/scripts/count

```
#!/bin/bash
echo $#
```

File: /home/alice/scripts/name

```
#!/bin/bash
read -p "What is your name?!" name
echo "Your name has `./count $name` words."
```

Converting the last example into a function

File: /home/alice/scripts/name2

```
#!/bin/bash
count()
{
  echo $#
}

read -p "What is your name?!" name
echo "Your name has ‘count $name‘ words."
```

Freaky tricks with scripts and functions

- You can use redirection with scripts
- You can use redirection with functions
- You can use scripts within pipes
- You can use functions within pipes

Freaky tricks with scripts and functions

- You can use redirection with scripts
- You can use redirection with functions
- You can use scripts within pipes
- You can use functions within pipes
- Anything you can do in a script, you can do interactively

Freaky tricks with scripts and functions

- You can use redirection with scripts
- You can use redirection with functions
- You can use scripts within pipes
- You can use functions within pipes
- Anything you can do in a script, you can do interactively

```
prompt$
```

Freaky tricks with scripts and functions

- You can use redirection with scripts
- You can use redirection with functions
- You can use scripts within pipes
- You can use functions within pipes
- Anything you can do in a script, you can do interactively

```
prompt$ count() {
```

Freaky tricks with scripts and functions

- You can use redirection with scripts
- You can use redirection with functions
- You can use scripts within pipes
- You can use functions within pipes
- Anything you can do in a script, you can do interactively

```
prompt$ count() {
> 
```

Freaky tricks with scripts and functions

- You can use redirection with scripts
- You can use redirection with functions
- You can use scripts within pipes
- You can use functions within pipes
- Anything you can do in a script, you can do interactively

```
prompt$ count() {
> echo $#
```

Freaky tricks with scripts and functions

- You can use redirection with scripts
- You can use redirection with functions
- You can use scripts within pipes
- You can use functions within pipes
- Anything you can do in a script, you can do interactively

```
prompt$ count() {
> echo $#
> 
```

Freaky tricks with scripts and functions

- You can use redirection with scripts
- You can use redirection with functions
- You can use scripts within pipes
- You can use functions within pipes
- Anything you can do in a script, you can do interactively

```
prompt$ count() {
> echo $#
> }
```

Freaky tricks with scripts and functions

- You can use redirection with scripts
- You can use redirection with functions
- You can use scripts within pipes
- You can use functions within pipes
- Anything you can do in a script, you can do interactively

```
prompt$ count() {
> echo $#
> }
prompt$
```

Freaky tricks with scripts and functions

- You can use redirection with scripts
- You can use redirection with functions
- You can use scripts within pipes
- You can use functions within pipes
- Anything you can do in a script, you can do interactively

```
prompt$ count() {
> echo $#
> }
prompt$ count Does this really work
```

Freaky tricks with scripts and functions

- You can use redirection with scripts
- You can use redirection with functions
- You can use scripts within pipes
- You can use functions within pipes
- Anything you can do in a script, you can do interactively

```
prompt$ count() {
> echo $#
> }
prompt$ count Does this really work
4
prompt$
```

How can I see what functions are declared in my shell?

`declare -f`

- Shows declared functions
- Drop the "`-f`" to see *everything* declared

```
prompt$
```

How can I see what functions are declared in my shell?

`declare -f`

- Shows declared functions
- Drop the "`-f`" to see *everything* declared

```
prompt$ declare -f
```

How can I see what functions are declared in my shell?

`declare -f`

- Shows declared functions
- Drop the "`-f`" to see *everything* declared

```
prompt$ declare -f
count ()
{
    echo $#
}
prompt$
```

Including other files

`source file`

- Causes `bash` to read (and execute) `file`
- Shorthand: "`. file`"

- Idea: declare useful functions in a separate file
- Then: "insert" that file into our current one
- Just like `#include` in C
- Allows us to easily re-use functions in several scripts

Terminating a shell script

- The `exit` command will cause a shell script to terminate
  - Even when used within a shell function
- Use `exit` *n* to specify the return code of the process
  - Which process?
  - The shell executing your shell script, of course

Terminating a shell function

- The `return` command will cause a shell function to terminate
- Use `return` *n* to specify the return code
  - Wait, does a function call start a new process?
  - Sometimes — it depends how it is called

Determining the status code

`$?` : Gives the status code of the last "command"

- Could be a shell builtin
- Could be a shell function
- Could be the last process that ran

Shell script: code1

```
#!/bin/bash
strange()
{
  echo "Strange function"
  return 42
}

ret=$(strange)
echo "Got: $ret"
echo "Return code: $?"
exit 3
```

```
prompt$
```

Shell script: code1

```
#!/bin/bash
strange()
{
  echo "Strange function"
  return 42
}

ret=$(strange)
echo "Got: $ret"
echo "Return code: $?"
exit 3
```

```
prompt$ ./code1
```

Shell script: code1

```
#!/bin/bash
strange()
{
  echo "Strange function"
  return 42
}

ret=$(strange)
echo "Got: $ret"
echo "Return code: $?"
exit 3
```

```
prompt$ ./code1
Got: Strange function
Return code: 0
prompt$
```

Shell script: code1

```
#!/bin/bash
strange()
{
  echo "Strange function"
  return 42
}

ret=$(strange)
echo "Got: $ret"
echo "Return code: $?"
exit 3
```

```
prompt$ ./code1
Got: Strange function
Return code: 0
prompt$ echo $?
```

Shell script: code1

```
#!/bin/bash
strange()
{
  echo "Strange function"
  return 42
}

ret=$(strange)
echo "Got: $ret"
echo "Return code: $?"
exit 3
```

```
Return code: 0
prompt$ echo $?
3
prompt$
```

Shell script: code2

```
#!/bin/bash
strange()
{
  echo "Strange function"
  return 42
}

ret=$(strange)
echo "Return code: $?"
echo "Got: $ret"
exit 3
```

```
prompt$
```

Shell script: code2

```
#!/bin/bash
strange()
{
  echo "Strange function"
  return 42
}

ret=$(strange)
echo "Return code: $?"
echo "Got: $ret"
exit 3
```

```
prompt$ ./code2
```

Shell script: code2

```
#!/bin/bash
strange()
{
  echo "Strange function"
  return 42
}

ret=$(strange)
echo "Return code: $?"
echo "Got: $ret"
exit 3
```

```
prompt$ ./code2
Return code: 42
Got: Strange function
prompt$
```

Shell script: code2

```
#!/bin/bash
strange()
{
  echo "Strange function"
  return 42
}

ret=$(strange)
echo "Return code: $?"
echo "Got: $ret"
exit 3
```

```
prompt$ ./code2
Return code: 42
Got: Strange function
prompt$ echo $?
```

Shell script: code2

```
#!/bin/bash
strange()
{
  echo "Strange function"
  return 42
}

ret=$(strange)
echo "Return code: $?"
echo "Got: $ret"
exit 3
```

```
Got: Strange function
prompt$ echo $?
3
prompt$
```

`declare` : see what's declared in the shell

`exit` *n* : exit the shell, specify status code *n*

`mktemp` : create a temporary file

`printf` : print things, like `printf` in C

`return` *n* : exit a shell function, specify status code *n*

`source` : read another file into the shell

`which` : determine which executable runs

An appropriate xkcd comic: http://xkcd.com/1654

End of lecture