Course Evaluations

Please fill them out! All feedback is welcome and appreciated!
Announcements

• Extra office hours
  • Tuesday, August 4, 2-4pm (BA3201)
  • Friday, August 7, 12:30-3pm (BA3289)
  • Monday, August 10, 1-4pm (BA3201)
• Labs are open and staffed tonight:
  • BA2210 and BA2220
• A4 is due next Friday, August 7 by 11:59pm
Tutorial Ideas

• Get your questions answered

• Discuss A4

• Work on *Suggested Exercises* from week 9 onward:
  

• If you haven’t already been doing so, practise your code reading comprehension skills by reading all of the lecture examples
Cloning the Course Website

$ git clone https://github.com/...
$ cd csc209-summer-2015
$ cd lectures/weekN
$ ./run.sh example.c

$ git pull
Agenda

• Shell Scripting
Shell Scripting

See http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html for a more detailed Bash tutorial
Scripting Languages
Programming vs Scripting Languages

C
C++
Java

JavaScript
Ruby
Python

- Compiled
- High(er) runtime performance
- More difficult to write
- “Serious”

- Interpreted
- Slower performance
- Easy to write
- “Toy”
Programming vs Scripting Languages
Shell as an Interactive Application

- Use *interactively* as our command interpreter
- Use for day-to-day tasks
  - Manipulating files and directories
  - Various pre-installed programs and our own
Shell as an Scripting Language

• Bundle up a pre-existing sequence of commands into a (shell) script file

• Begin the file with the shebang line and set the file to be executable with chmod u+x

  • `#!/bin/bash`

• Other command interpreters available (`/bin/sh`, `/bin/zsh`, your own `.sh209`), and you can use this trick with other scripting languages (i.e. `/usr/local/bin/bin/python`)
Builtins

- As we saw in A3, some shell commands are *built-in* and do not need to *fork+exec* to run
  - `cd`
  - `exit`
  - `echo`
  - `read`
  - `test`
  - `shift`
  - `wait`
  - `set`
  - `unset`
  - `export`
  - `expr`

See `man bash-builtin` to learn about these and many more.
String-ly typed

The only data type is string
Program Arguments

• Recall that your sh209 implementation made a call to `execvp` and passed in an `argv` array:

```

$ ./myprog foo bar 209
```

```
argv[0]   argv[2]
```
$ ./argv foo bar 209
argv[0] = "./argv"
argv[1] = "foo"
argv[2] = "bar"
argv[3] = "209"
Environment Variables

• We’ve seen a few of these before: USER, HOME, PATH

• Inspect their value as $ variables:

  $ echo $USER
  g5stdnt
Environment Variables

• What happens if we pass a variable in to one of our programs?

    $ ./myprog $USER
String Replacement

• The shell *substitutes* or expands expressions (like $ variables) through string replacement *before* execution:

\[
\text{
$\text{echo } \$\text{USER}$
}\rightarrow
\text{
$\text{echo g5stdnt}$
}
\]
Environment Variables

- **env**: print the environment (also runs programs in a modified environment)

  ```
  $ env
  
  USER=g5stdnt
  HOME=/h/u15/c5/00/g5stdnt
  PATH=/bin:/usr/bin:...
  
  ...
Environment Variables

• You can create new environment variables or assign new values using the `export` builtin:

```bash
$ export FOO=bar

$ echo $FOO
bar

$ export FOO=209

$ echo $FOO
209
```
Environment Variables

• You can remove an environment variable using `unset`:

```
$ export FOO=bar
$ echo FOO is $FOO
FOO is bar
$ unset FOO
$ echo FOO is $FOO
FOO is
```
Environment Variables

- Child processes also *inherit* the environment mapping
- C programs can access it using the `getenv()` call

`getenv.c`
The true signature of the main function actually has a third argument `envp`, a `NULL`-terminated array of `KEY=VALUE` pairs from the shell environment!
Wildcard Substitution

- What arguments does the program get when you run the following?

$ ./myprog *.c

(these are called *glob expressions*)
Local Variables

- Different than environment variables, but are $ substituted in the same way
  - But unlike environment variables, child processes do not see them

- Assignment:
  
  $ name=value

- Single assignment:
  
  $ BIN209="/u/csc209h/summer/pub/bin"

  $ echo $BIN209

- Lists of values are typically space separated:
  
  $ STAFF="pdm t5champ"
Local Variables

- The shell will perform substitutions anywhere it sees them, including in assignments!

```
$ REPO=${HOME}/csc209-summer-2015

$ ALL_SRC=*.c
```
Quoting

- You can use single ('), double ("") or backtick (``) quotes to change how the shell performs substitutions.
- **Double quotes** inhibit wildcard replacement only.
  
  ```
  $ echo "*.c"
  
  *.c
  ```

- **Single quotes** inhibit wildcard replacement, variable substitution and command substitution.
  
  ```
  $ echo '\$USER *.c `ls`'
  
  $USER *.c `ls`
  ```
Quoting — Command Substitution

• *Backtick quotes* perform command substitution, capturing the standard output of a program as a string:

```bash
$ FOO=`echo hello world`

$ echo $FOO

hello world
```

*The backtick quote is on the same key as ~, beside 1*
Quoting — Command Substitution

- $(...) is an alternate syntax for backticks

```
$ LS_OUT=$(ls *.c)

$ echo $LS_OUT

argv.c getenv.c envp.c ...
```
Quoting — Examples

• Easy to make mistakes and over-quote or under-quote!

$ echo Today is date
Today is date

$ echo Today is `date`
Today is Thu Jul 30 14:13:11 EDT 2015

$ echo "Today is `date`"
Today is Thu Jul 30 14:13:11 EDT 2015

$ echo ’Today is `date`’
Today is `date`
Quoting — Practise

• What do the following statements produce if the current directory contains the following non-executable files?

   a  b  c

   $ echo *

   $ echo ls *

   $ echo \ls *

   $ echo "ls *"

   $ echo 'ls *

   $ echo `ls *`

   $ echo `*`
Control Statement — for

• Iterate through a space separated list

```bash
for colour in red green blue orange; do
echo My favourite is $colour
done
```

My favourite is red
My favourite is green
My favourite is blue
My favourite is orange
Control Statement — for

- Remember the quoting rules...

```bash
for colour in "red green blue orange"; do
echo My favourite is $colour
done
```

My favourite is red green blue orange
Control Statement — for

• Remember the quoting rules...

```bash
COLOURS="red green blue orange"
for colour in $COLOURS; do
    echo My favourite is $colour
done
```

My favourite is red
My favourite is green
My favourite is blue
My favourite is orange
Control Statement — for

• Remember the quoting rules...

```
COLOURS="red green blue orange"
for colour in "$COLOURS"; do
echo My favourite is $colour
done
```

My favourite is red green blue orange
Exit Status Code

- Recall that C programs can terminate by either calling `exit(status)` or by returning a `status` from `main()`

- Convention: an exit status of 0 indicates success, and anything else is a failure

  - Think Boolean for failed (so 1, which is true in C, indicates that there was a failure, while 0 is false, there was not a failure, hence it was a success)
Exit Status Code

• In the shell, special variable $? contains the exit status of the last command executed:

```bash
$ echo Hello!
Hello!
```

```bash
$ echo $?
0
```

• Apparently, echo exited successfully!
Exit Status Code

- There are two simple programs which do nothing except return a status code

- **true**: do nothing, *successfully*
  
  ```
  $ true
  
  $ echo $? 
  0
  ```

- **false**: do nothing, *unsuccessfully*
  
  ```
  $ false
  
  $ echo $? 
  1
  ```
Shell operators corresponding to Boolean AND/OR that *short-circuit* (like in C, Java and Python!) depending on the exit status codes:

```bash
$ false && false; echo $? 
1

$ false && true; echo $? 
1

$ true && false; echo $? 
1

$ true && true; echo $? 
0
```
Shell operators corresponding to Boolean AND/OR that either succeed or fail depending on the success or failure (i.e. the exit status codes) of the operands

```bash
$ false || false; echo $?
1

$ false || true; echo $?
0

$ true || false; echo $?
0

$ true || true; echo $?
0
```
Shell operators corresponding to Boolean AND/OR that either succeed or fail depending on the success or failure (i.e. the exit status codes) of the operands

$ true && echo always
always

$ echo $?
0

$ true || echo never
$ echo $? 
0
These operators *short-circuit* (like they do C, Java and Python) meaning they may not have to execute the second argument.

```bash
$ false && echo never

$ echo $? 
1

$ false || echo always

always

$ echo $? 
0
```
These operators *short-circuit* (like they do C, Java and Python) meaning they may not have to execute the second argument

```
$ echo first && echo second
first
second
$ echo $?
0

$ echo first || echo second
first
$ echo $?
0
```
Control Statement — if

```bash
if condition; then
    # condition succeeded (status 0)
    ...
else
    # condition failed (status != 0)
    ...
fi
```

- **condition** is a command that is executed and its exit status is checked for success/failure
Built-in `test`

- Construct conditional statements, returning a suitable exit status
- Also usable as `[[ ... ]]`

```
if test ...; then
  ...
fi
```
```
if [[ ... ]]; then
  ...
fi
```
## Builtin test

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-d filename</code></td>
<td>Exists as a directory</td>
</tr>
<tr>
<td><code>-f filename</code></td>
<td>Exists as a regular file</td>
</tr>
<tr>
<td><code>-r filename</code></td>
<td>Exists as a readable file</td>
</tr>
<tr>
<td><code>-w filename</code></td>
<td>Exists as a writable file</td>
</tr>
<tr>
<td><code>-x filename</code></td>
<td>Exists as an executable file</td>
</tr>
<tr>
<td><code>-z string</code></td>
<td>True if empty string</td>
</tr>
<tr>
<td><code>str1 = str2</code></td>
<td>True if str1 equals str2</td>
</tr>
<tr>
<td><code>str1 != str2</code></td>
<td>True if str1 not equal to str2</td>
</tr>
<tr>
<td><code>int1 -eq int2</code></td>
<td>True if int1 equals int2</td>
</tr>
<tr>
<td><code>-ne -gt -lt -le -ge</code></td>
<td>Not equal, greater than, less than, less than or equal, greater than or equal</td>
</tr>
<tr>
<td><code>-a -o</code></td>
<td>And, or</td>
</tr>
</tbody>
</table>
Built-in `test` — Examples

```bash
if test -f run.sh; then
    echo "A file named run.sh exists"
fi

if [[ -x run.sh ]]; then
    echo "run.sh exists and is executable"
fi

# Ensure directory "bin" exists
[[ ! -d bin ]] && mkdir bin
```
Control Statement — while

• Loops as long as \textit{condition} succeeds

\begin{verbatim}
while condition; do
  ...
done
\end{verbatim}
Positional Parameters

• The shell gives us *special* builtin variables that correspond to the arguments passed into a shell script:

```
$ ./script.sh foo bar 209
```

$0  $1  $2  $3
Positional Parameters

posargs.sh:

```bash
#!/bin/bash

echo arg0: $0
echo arg1: $1
echo arg2: $2
echo all: $*
```

Don’t forget to `chmod u+x` to make your script executable!
## Positional Parameters

<table>
<thead>
<tr>
<th>What it references</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$0</td>
<td>Name of the script</td>
</tr>
<tr>
<td>$#</td>
<td>Number of positional parameters</td>
</tr>
<tr>
<td>$*</td>
<td>Lists all positional parameters</td>
</tr>
<tr>
<td>$@</td>
<td>Same as $* except when in quotes</td>
</tr>
<tr>
<td>&quot;$*&quot;</td>
<td>Expands to a single argument (&quot;$1 $2 $3&quot;)</td>
</tr>
<tr>
<td>&quot;$@&quot;</td>
<td>Expands to separate arguments (&quot;$1&quot; &quot;$2&quot; &quot;$3&quot;)</td>
</tr>
<tr>
<td>$1 .. $9</td>
<td>First 9 positional parameters</td>
</tr>
<tr>
<td>${10}</td>
<td>10th positional parameter</td>
</tr>
</tbody>
</table>
Looping over Quoted Positional Parameters

- Correctly quoting in a `for` loop can be tricky!

```bash
for ARG in "$*"; do
echo "$ARG"
done
```

Quotes mean arguments are all in one string.

```bash
for ARG in $*; do
echo "$ARG"
done
```

One element for each argument (irrespective of how it was passed)
Looping over Quoted Positional Parameters

for ARG in "@$"; do
echo "$ARG"
done

Quotes in original argument list are preserved

for ARG in $@; do
echo "$ARG"
done

Same as $*, does not preserve original quotes
loopquotes.sh
• **set**: assigns positional parameters to its arguments

```
$ set foo 209 BAZ

$ echo $3 $2 $1

BAZ 209 foo
```

• Useful in conjunction with command expansion:

```
$ set `date`

$ echo "Today is $2 $3, $6"

Today is Jul 30, 2015
shift

- **shift**: throws away $0$ and assigns $0 \leftarrow 1$, then $1 \leftarrow 2$, etc.

```bash
$ set first second third

$ echo $1 $2 $3

first second third

$ shift

$ echo $1 $2 $3

second third

$ shift

$ echo $1 $2 $3

third
```
• Use `shift` with a `while` loop to handle a variable number of arguments to your script:

```
#!/bin/bash

while [[ "$1" ]]; do
  echo ARG: "$1"
  shift
done
```

```
$ ./peel.sh a "b c" d
ARG: a
ARG: b c
ARG: d
```
run.sh
expr — evaluate expression

• Since the shell only deals with strings, to perform arithmetic we need some extra help:

```
$ x=1

$ x=`expr $x + 1`

$ y=`expr 3 \* 5`

$ echo $x $y

2 15
```
expr — evaluate expression

• *expr* can also check if one string is a *substring* of another

```bash
$ expr "Hello World" : Hello
5
$ expr "Hello World" : Bonjour
0
```

• This is very weak! Use *awk/sed* or *Python* if you need more
read — consume standard input

- **read** will read one line from standard input and assigns successive space-separated words to the specified variables.
- Leftover words are assigned to the last variable.

```bash
#!/bin/bash
echo "Enter name: "
read FIRST LAST
echo "First: $FIRST"
echo "Last : $LAST"
```

```
$ ./name.sh
Enter name: A B C
First: A
Last : B C
```
prefixcat.sh
Subroutines

• You can create your own functions or subroutines:

```bash
myfunc() {
    arg1=$1
    arg2=$2
    echo $arg1 $globalvar
    return 0
}
```

• `globalvar="I am global"
• `myfunc num1 num2`
Defining Subroutines

- The shell lets you group code into reusable subroutines or functions.

```bash
#!/bin/bash

helper() {
    echo "$0"
    echo "$@
    return 10
}

helper a "B C" d

echo $?
```
Next Week

• Extra office hours: Tuesday and Friday (plus a week Monday)

• A4 due on Friday

• Next Thursday’s lecture will be exam review
  • Bring your questions
Labs

• Go ask questions and get help!

• **BA2210** and **BA2220**