CS2211b

Software Tools and Systems Programming

Week 6a

Shell Scripts: Part 3

To complete your Midterm Check-In, please visit:

feedback.uwo.ca

your Feedback

Western Science

Western

Announcements

Lab 5 on OWL

Week 4 and 5 Q&A Posted

No Office Hours or Labs on Reading Week

Quiz #2 Next Lecture

Change in Assignment Schedule:

Assignment #	To Be Posted On	Due On	Due in
1	January 17th	January 31st	2 weeks
2	January 31st	February 17th	2.5 weeks
3	February 28th	March 14th	2 weeks
4	March 21st	April 4th	2 weeks

Week 5b Review

- Arithmetic:
  - `expr`
  - `bc`
  - `(( ))`
  - `$(( ))`
- Arguments:
  - `$0`
  - `$1 - $9`
  - `$*`
  - `$#`
  - `shift`
- If statement:

```
if command1; then
   commands_to_run_if_command1_successful
elif command2; then
   commands_to_run_if_command2_successful
else
   commands_to_run_otherwise
fi
```

Week 5b Review

- `test`:
  - `test expr1 -opt expr2` - `[ expr1 -opt expr2 ]`
  - `test -opt expr` - `[ -opt expr ]`
- And/Or:
  - `&&`
  - `||`
- For loop:

```
for var in value1 value2 …; do
    commands_to_run
done
```

- While loop:

```
while command; do
    commands_to_run
done
```

Shell Scripts: Part 3

Case Statements

- `case` statements support branching based on the value of a string and a set of patterns.
- If the string matches a pattern, that block of code is executed.
- Patterns are a mix of shell wild cards and regex.
- **Syntax:**

```
case string in
    pattern1)
        commands_to_run_if_pattern1_matched
        ;;
    pattern2)
        commands_to_run_if_pattern2_matched
        ;;
    …
esac
```

Case Statements

Example 1:

/cs2211/week6/ex1.sh

```
#!/bin/bash
read -p 'Choose command [1-4]: ' reply
echo
case $reply in
  "1")
    date
    ;;
  "2"|"3")
    pwd
    ;;
  "4")
    ls
    ;;
  *)
    echo 'Illegal choice!'
    ;;
esac
```

Case Statements

Example 1:

/cs2211/week6/ex1.sh

```
#!/bin/bash
read -p 'Choose command [1-4]: ' reply
echo
case $reply in
  "1")
    date
    ;;
  "2"|"3")
    pwd
    ;;
  "4")
    ls
    ;;
  *)
    echo 'Illegal choice!'
    ;;
esac
```

Matches the literal string “1”

Matches the literal string “4”

Case Statements

Example 6:

/cs2211/week6/ex1.sh

```
#!/bin/bash
read -p 'Choose command [1-4]: ' reply
echo
case $reply in
  "1")
    date
    ;;
  "2"|"3")
    pwd
    ;;
  "4")
    ls
    ;;
  *)
    echo 'Illegal choice!'
    ;;
esac
```

Matches the literal string “2” or “3”.

As in regular expression the | metacharacter is for OR

Case Statements

Example 1:

/cs2211/week6/ex1.sh

```
#!/bin/bash
read -p 'Choose command [1-4]: ' reply
echo
case $reply in
  "1")
    date
    ;;
  "2"|"3")
    pwd
    ;;
  "4")
    ls
    ;;
  *)
    echo 'Illegal
    ;;
esac
```

Matches anything. Like the shell wildcard *

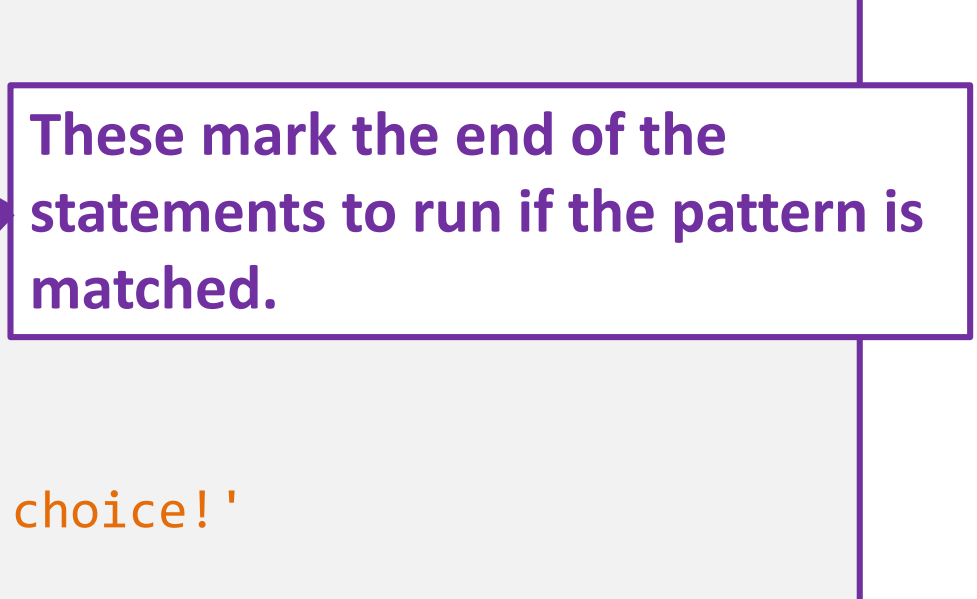
Pattern matches are done in order, so this will only match if the above patterns do not.

Case Statements

Example 1:

/cs2211/week6/ex1.sh

```
#!/bin/bash
read -p 'Choose command [1-4]: ' reply
echo
case $reply in
  "1")
    date
    ;;
  "2"|"3")
    pwd
    ;;
  "4")
    ls
    ;;
  *)
    echo 'Illegal choice!'
    ;;
esac
```



Case Statements

Example 1:

/cs2211/week6/ex1b.sh

```
#!/bin/bash
read -p 'Choose command [1-4]: ' reply
echo
case $reply in
  "1")
    date;;
  "2"|"3")
    pwd;;
  "4")
    ls;;
  *)
    echo 'Illegal choice!';;
esac
```

The ;; can go on the same line as a command as shown here.

Case Statements

Example 1:

/cs2211/week6/ex1c.sh

```
#!/bin/bash
read -p 'Choose command [1-4]: ' reply
echo
case $reply in
  "1")
    who
    date;;
  "2"|"3")
    pwd;;
  "4")
    ls;;
  *)
    echo 'Illegal choice!';;
esac
```

You can have multiple commands per pattern.

In this case both who and date would be run if a 1 is input.

Case Statements

Example 2:

/cs2211/week6/ex2.sh

```
#!/bin/bash

for file in $*; do
    if [ -f $file ]; then
        case $file in
            *.txt) echo "$file is a text file.";;
            *.sh|*.bash) echo "$file is a shell script.";;
            *.[cC]) echo "$file is a c source code file.";;
            *.[oO]) echo "$file is an object file.";;
            *.htm|*.html) echo "$file is a HTML file.";;
            *) echo "$file is an unknown file type.";;
        esac
    fi
done
```

Case Statements

Example 2:

/cs2211/week6/ex2.sh

```
#!/bin/bash

for file in $*; do
    if [ -f $file ]; then
        case $file in
            *.txt) echo "$fil
            *.sh|*.bash) echo "$file is a shell script.";;
            *.[cC]) echo "$file is a c source code file.";;
            *.[oO]) echo "$file is an object file.";;
            *.htm|*.html) echo "$file is a HTML file.";;
            *) echo "$file is an unknown file type.";;
        esac
    fi
done
```

Loops through each argument.

Assumes each argument does not contain a space. Would not work correctly if they did.

Bad practice. Don't do this.

Case Statements

Example 2:

/cs2211/week6/ex2.sh

```
#!/bin/bash

for file in $*; do
    if [ -f $file ]; then
        case $file in
            *.txt) echo "$file is a text file.";;
            *.sh|*.bash) echo "$file is a shell script.";;
            *.[cC]) echo "$file is a c source code file.";;
            *.[oO]) echo "$file is an object file.";;
            *.htm|*.html) echo "$file is a HTML file.";;
            *) echo "$file is an unknown file type.";;
        esac
    fi
done
```

Checks if the current argument is a regular file.

Case Statements

Example 2:

/cs2211/week6/ex2.sh

```
#!/bin/bash

for file in $*; do
    if [ -f $file ]; then
        case $file in
            *.txt) echo "$file is a text file.";;
            *.sh|*.bash) echo "$file is a shell script.";;
            *.[cC]) echo "$file is a c source code file.";;
            *.[oO]) echo "$file is an object file.";;
            *.htm|*.html) echo "$file is a HTML file.";;
            *) echo "$file is an unknown file type.";;
        esac
    fi
done
```

Patterns for checking the file extension. Very similar to shell wild cards.

Case Statements

Example 2:

/cs2211/week6/ex2.sh

```
#!/bin/bash

for file in $*; do
    if [ -f $file ]; then
        case $file in
            *.txt) echo "$file is a text file.";;
            *.sh|*.bash) echo "$file is a shell script.";;
            *.[cC]) echo "$file is a c source code file.";;
            *.[oO]) echo "$file is an object file.";;
            *.htm|*.html) echo "$file is a HTML file.";;
            *) echo "$file is an unknown file type.";;
        esac
    fi
done
```

Print the file name and the file type based on the file extension.

Case Statements

Example 2:

Better solution using while loop.

/cs2211/week6/ex2b.sh

```
#!/bin/bash

while [ $# -gt 0 ]; do
    if [ -f $1 ]; then
        case $1 in
            *.txt) echo "$1 is a text file.";;
            *.sh|*.bash) echo "$1 is a shell script.";;
            *.[cC]) echo "$1 is a c source code file.";;
            *.[oO]) echo "$1 is an object file.";;
            *.htm|*.html) echo "$1 is a HTML file.";;
            *) echo "$1 is an unknown file type.";;
        esac
    fi
    shift
done
```

Case Statements

Example 2:

Better solution using for loop.

/cs2211/week6/ex2c.sh

```
#!/bin/bash

for file in "$@"; do
    if [ -f $file ]; then
        case $file in
            *.txt) echo "$file is a text file.";;
            *.sh|*.bash) echo "$file is a shell script.";;
            *.[cC]) echo "$file is a c source code file.";;
            *.[oO]) echo "$file is an object file.";;
            *.htm|*.html) echo "$file is a HTML file.";;
            *) echo "$file is an unknown file type.";;
        esac
    fi
done
```

Case Statements

Example 2:

Better solution using for loop.

/cs2211/week6/ex2c.sh

```
#!/bin/bash

for file in "$@"; do
    if [ -f $file ]; th
        case $file in
            *.txt) echo
            *.sh|*.bash
            *.[cC]) ech
            *.[oO]) echo "$file is an object file.";;
            *.htm|*.html) echo "$file is a HTML file.";;
            *) echo "$file is an unknown file type.";;
        esac
    fi
done
```

$@ returns all arguments in an array. "$@" ensures that arguments with spaces are treated as one value for the loop.

Case Statements

Example 2:

Better solution using for loop.

/cs2211/week6/ex2c.sh

```
#!/bin/bash

for file in "$@"; do
    if [ -f $file ]; th
        case $file in
            *.txt) echo
            *.sh|*.bash
            *.[cC]) ech
            *.[oO]) echo "$file is an object file.";;
            *.htm|*.html) echo "$file is a HTML file.";;
            *) echo "$file is an unknown file type.";;
        esac
    fi
done
```

This is what a for loop does by default if we give no values. So we could just omit this and the `in`.

Case Statements

Example 2:

/cs2211/week6/ex2d.sh

Better solution using for loop.

Shorter version.

```
#!/bin/bash

for file; do
    if [ -f $file ]; then
        case $file in
            *.txt) echo "$file is a text file.";;
            *.sh|*.bash) echo "$file is a shell script.";;
            *.[cC]) echo "$file is a c source code file.";;
            *.[oO]) echo "$file is an object file.";;
            *.htm|*.html) echo "$file is a HTML file.";;
            *) echo "$file is an unknown file type.";;
        esac
    fi
done
```

Using `read` with `while`

- We can use the `read` command with the while loop to read multiple lines of input.
- **Syntax:**

```
while read vars …; do
   commands_to_run
done
```

- Will keep reading input until end of file (EOF) is given (Ctrl-D).
- We can use this to make our own filter.

Using `read` with `while`

Example 3: Make a filter that removes everything but numbers on each line.

Using `read` with `while`

Example 3: Make a filter that removes everything but numbers on each line.

/cs2211/week6/ex3.sh

```
#!/bin/bash

while read line; do
        nums=`echo -n "$line" | tr -cs '0-9' ' '`
        nums=`echo -n $nums`

        if [ ! -z "$nums" ]; then
                echo $nums
        fi
done
```

Using `read` with `while`

Example 3: Make a filter that removes everything but numbers on each line.

/cs2211/week6/ex3.sh

```
#!/bin/bash

while read line; do
        nums=`echo -n "$line" | tr -cs '0-9' ' '`
        nums=`echo -n $nums`

        if [ ! -z "$nums" ]; then
                echo $nums
        fi
done
```

Read a whole line into the $line shell variable.

Keep reading lines until EOF is hit.

Using `read` with `while`

Example 3: Make a filter that removes everything but numbers on each line.

/cs2211/week6/ex3.sh

```
#!/bin/bash

while read line; do
        nums=`echo -n "$line" | tr -cs '0-9' ' '`
        nums=`echo -n $nums`

        if [ ! -z "$nums" ]; then
                echo $nums
        fi
done
```

Use `tr` command to replace everything but numbers with a space.

Using `read` with `while`

Example 3: Make a filter that removes everything but numbers on each line.

/cs2211/week6/ex3.sh

```
#!/bin/bash

while read line; do
        nums=`echo -n "$line" | tr -cs '0-9' ' ' `
        nums=`echo -n $nums`

        if [ ! -z "$nums" ]; then
                echo $nums
        fi
done
```

Store result in $nums.

Using read with while

Example 3: Make a fil
numbers on each line

Trick to trim white space.

As we did not use quotes around $nums, echo removes spaces between arguments. -n ensures that echo does not add a linebreak.

/cs2211/week6/ex3.sh

```
#!/bin/bash

while read line; do
        nums=`echo -n "$line" | tr -cs '0-9' ' ' `
        nums=`echo -n $nums`

        if [ ! -z "$nums" ]; then
                echo $nums
        fi
done
```

Using `read` with `while`

Example 3: Make a filter that removes everything but numbers on each line.

/cs2211/week6/ex3.sh

```
#!/bin/bash

while read line; do
        nums=`echo -n "
        nums=`echo -n $

        if [ ! -z "$nums" ]; then
                echo $nums
        fi
done
```

Check if the line is empty (there was no numbers on the line). If this is the case, nums should now be an empty string.

If the line is not empty, output the numbers.

Using `read` with `while`

Example 3: Test with manual input.

```
[dservos5@cs2211b week6]$ ex3.sh
This is a test with 1 number.
1
This is a 2nd test with 2 numbers.
2 2
This line has no numbers!
1.23 + 4.32 = 5.55
1 23 4 32 5 55
```

Input (typed in)

Output (from script)

Ctrl-D to stop

Using `read` with `while`

Example 3: Test with redirection.

```
[dservos5@cs2211b week6]$ cat textlines.txt
Section 6.9 makes use of this feature, while some
situations are presented in Chapter 13 (featuring shell
programming).

The system's default then applies (666 for files and 777
for directories).

Line with no numbers!

Line with 1 number.

[dservos5@cs2211b week6]$ ex3.sh < textlines.txt
6 9 13
666 777
1
```

Using `read` with `while`

Example 3: Test with pipe.

```
[dservos5@cs2211b week6]$ who
rgabrie  pts/2     2018-02-12 13:27 (nexus-5.wireless.uwo.ca)
dservos5 pts/3     2018-02-12 12:17 (135-23-234-30.cpe.pppoe.ca)


[dservos5@cs2211b week6]$ who | ex3.sh
2 2018 02 12 13 27 5
5 3 2018 02 12 12 17 135 23 234 30
```

Using `read` with `while`

Example 4: The file input.txt contains a number of lines, each containing exactly 3 integers separated by spaces. Write a script to add the numbers on each line.

Using `read` with `while`

Example 4: The file input.txt contains a number of lines, each containing exactly 3 integers separated by spaces. Write a script to add the numbers on each line.

/cs2211/week6/ex4.sh

```
#!/bin/bash

while read a b c; do
        sum=$((a + b + c))
        echo $sum
done < input.txt
```

Using `read` with `while`

Example 4: The file input.txt contains a number of lines, each containing exactly 3 integers separated by spaces. Write a script to add the numbers on each line.

/cs2211/week6/ex4.sh

```
#!/bin/bash

while read a b c; do
        sum=$((a + b + c))
        echo $sum
done < input.txt
```

Read in three values into $a, $b and $c.

Using `read` with `while`

Example 4: The file input.txt contains a number of lines, each containing exactly 3 integers separated by spaces. Write a script to add

/cs2211/week6/ex4.sh

```
#!/bin/bash

while read a b c; do
        sum=$((a + b
        echo $sum
done < input.txt
```

We can do redirection inside of the shell script.

This redirects the contents of input.txt in the current working directory into the while loop.

The loop is run once for each line, until EOF is encountered.

Using `read` with `while`

Example 4: Output.

```
[dservos5@cs2211b week6]$ cat input.txt
5 10 15
-10 20 3
1 1 1
1 2 3

[dservos5@cs2211b week6]$ ex4.sh
30
13
3
6
```

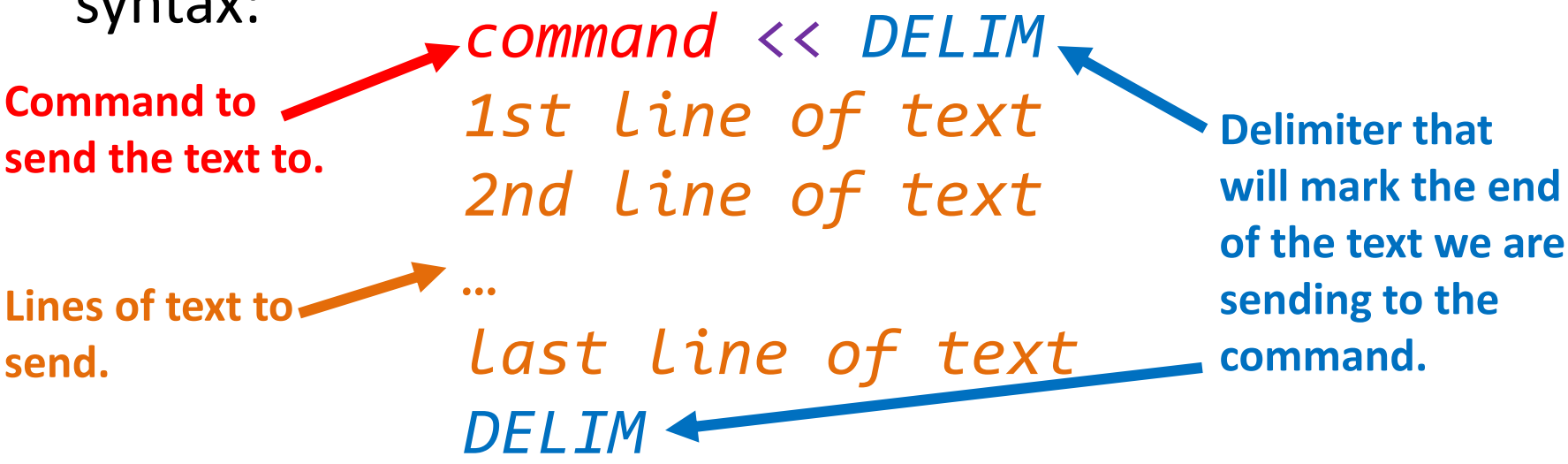
Here Document (<<)

- Sometimes we want to send text to the standard input of a command without storing it in a file or using another command like echo.
- The here document is another way we can send standard input to a command and uses the following syntax:

```
command << DELIM
1st line of text
2nd line of text
…
last line of text
DELIM
```

Here Document (<<)

- Sometimes we want to send text to the standard input of a command without storing it in a file or using another command like `echo`.
- The here document is another way we can send standard input to a command and uses the following syntax:



Here Document (<<)

Examples On the Command Line

```
[dservos5@cs2211b week6]$ wc << STOP
> This is a test!
> I can type text here with chars like * & ^ % $ # @ ! + - ) (
> So long as it is not the delimiter word it will keep taking input.
> STOP
   3   38 144

[dservos5@cs2211b week6]$ grep 'UNIX' << END
> The “user” category is served by the first 11 chapters, which is adequate
> for an introductory UNIX course. The “developer” is a shell or systems
> programmer who also needs to know how things work, say, how a directory is
> affected when a file is created or linked. For their benefit, the initial
> chapters contain special boxes that probe key concepts. This arrangement
> shouldn’t affect the beginner, who may quietly ignore these portions. UNIX
> shines through Chapters 16, 17, and 18, so these chapters are compulsory
> reading for systems programmers. END
> END
for an introductory UNIX course. The “developer” is a shell or systems
shouldn’t affect the beginner, who may quietly ignore these portions. UNIX
```

Here Document (<<)

Examples On the Command Line

Input to command.
Output of command.

```
[dservos5@cs2211b week6]$ wc << STOP
> This is a test!
> I can type text here with chars like * & ^ % $ # @ ! + - ) (
> So long as it is not the delimiter word it will keep taking input.
> STOP
   3   38 144

[dservos5@cs2211b week6]$ grep 'UNIX' << END
> The “user” category is served by the first 11 chapters, which is adequate
> for an introductory UNIX course. The “developer” is a shell or systems
> programmer who also needs to know how things work, say, how a directory is
> affected when a file is created or linked. For their benefit, the initial
> chapters contain special boxes that probe key concepts. This arrangement
> shouldn’t affect the beginner, who may quietly ignore these portions. UNIX
> shines through Chapters 16, 17, and 18, so these chapters are compulsory
> reading for systems programmers. END
> END
for an introductory UNIX course. The “developer” is a shell or systems
shouldn’t affect the beginner, who may quietly ignore these portions. UNIX
```

Here Document (<<)

Examples On the Command Line

```
[dservos5@cs2211b week6]$ wc << STOP
> This is a test!
> I can type text here with chars like * & ^ % $ # @ ! + - ) (
> So long as it is not the delimiter word it will keep taking input.
> STOP
   3   38 144

[dservos5@cs2211b week6]$ grep 'UNIX' << END
> The “user” category is served by the first 11 chapters, which is adequate
> for an introductory UNIX
> programmer who also needs
> affected when a file is
> chapters contain special
> shouldn’t affect the beginner, who may quietly ignore these portions. UNIX
> shines through Chapters 16, 17, and 18, so these chapters are compulsory
> reading for systems programmers. END
> END
for an introductory UNIX course. The “developer” is a shell or systems
shouldn’t affect the beginner, who may quietly ignore these portions. UNIX
```

Delimiter needs to be only thing on line to stop input.

Here Document (<<)

Example 5: In a shell script.

/cs2211/week6/ex5.sh

```
#!/bin/bash

cat << END_WELCOME
Hello $USER,
Welcome to my shell script.
Please input a word to search for:
END_WELCOME

read word
files=`grep -l "$word" * 2> /dev/null`

cat << END_FOUND

I found the following files in the
current working directory:
$files
END_FOUND
```

Here Document (<<)

Example 5: In a shell script.

/cs2211/week6/ex5.sh

```
#!/bin/bash

cat << END_WELCOME
Hello $USER,
Welcome to my shell script
Please input a word to sea
END_WELCOME

read word
files=`grep -l "$word" * 2

cat << END_FOUND

I found the following files in the
current working directory:
$files
END_FOUND
```

We can output the value of shell variables in here document text.

If we put quotes around the delimiter like:

```
cat << 'END_WELCOME'
```

The variables will be escaped (not expanded).

Here Document (<<)

Example 5: Output

```
[dservos5@cs2211b week6]$ ex5.sh
Hello dservos5,
Welcome to my shell script.
Please input a word to search for:
while

I found the following files in the
current working directory:
ex2b.sh
ex3.sh
ex4.sh
ex6.sh
textlines.txt
```

Here String (<<<)

- Like here document but sends the value of a string or shell variable to a command over standard input.
- Not part of POSIX and may not be available in all shells.
- **Syntax:**

```
command <<< "string"

command <<< $var
```

Here String (<<<)

Example 6:

/cs2211/week6/ex6.sh

```
#!/bin/bash

whoout=`who`

while read user pts login time host; do
        echo "$user:"
        echo "PTS: $pts"
        echo "Host: `tr -d '()' <<< $host`"
        echo "Groups`groups $user | grep -o ':.*$'`"
        echo "ID: `id $user`"
        echo
done <<< "$whoout"
```

Here String (<<<)

Example 6:

/cs2211/week6/ex6.sh

```
#!/bin/bash

whoout=`who`

while read user pts login time host; do
        echo "$user:"
        echo "PTS: $pts"
        echo "Host: `tr -d '()' <<< $host`"
        echo "Groups`groups $user | grep -o ':.*$'`"
        echo "ID: `id $user`"
        echo
done <<< "$whoout"
```

Loops through each line of the output of the who command and prints a bit more information about the user.

Here String (<<<)

Example 6:

/cs2211/week6/ex6.sh

```
#!/bin/bash

whoout=`who`

while read user pts login time host; do
        echo "$user:"
        echo "PTS: $pts"
        echo "Host: `tr -d '()' <<< $host`"
        echo "Groups`groups $user | grep -o ':.*$'`"
        echo "ID: `id $user`"
        echo
done <<< "$whoout"
```

Sends value of $host to `tr` command via standard input.

Removes the ()s around the hostname.

Here String (<<<)

Example 6:

/cs2211/week6/ex6.sh

```
#!/bin/bash

whoout=`who`

while read user pts l
        echo "$user:"
        echo "PTS: $p
        echo "Host: `tr -d '()' <<< $host`"
        echo "Groups`groups $user | grep -o ':.*$'`"
        echo "ID: `id $user`"
        echo
done <<< "$whoout"
```

Normally, the groups command also outputs the username again followed by a :

This only returns the groups. -o option to grep makes grep only return the match (not the whole line).

Here String (<<<)

Example 6:

/cs2211/week6/ex6.sh

```
#!/bin/bash

whoout=`who`

while read user pts login time host; do
        echo "$user
        echo "PTS: $
        echo "Host:
        echo "Groups`groups $user | grep -o ':.*$'`"
        echo "ID: `id $user`"
        echo
done <<< "$whoout"
```

The id command outputs user and group ids for a given user.

Here String (<<<)

Example 6: Output

```
[dservos5@cs2211b week6]$ who
nporrone pts/0        2018-02-12 14:44 (nexus-19.wireless.uwo.ca)
dservos5 pts/3        2018-02-12 12:17 (135-23-234-30.cpe.pppoe.ca)

[dservos5@cs2211b week6]$ ex6.sh
nporrone:
PTS: pts/0
Host: nexus-19.wireless.uwo.ca
Groups: nporrone gaulusers
ID: uid=410401009(nporrone) gid=410401009(nporrone)
groups=410401009(nporrone),410400001(gaulusers)

dservos5:
PTS: pts/3
Host: 135-23-234-30.cpe.pppoe.ca
Groups: grad gaulusers
ID: uid=17789(dservos5) gid=1985(grad)
groups=1985(grad),410400001(gaulusers)
```

Here String (<<<)

Example 6:

/cs2211/week6/ex6.sh

```
#!/bin/bash

whoout=`who`

while read user pts login time host; do
        echo "$user:"
        echo "PTS: $pts"
        echo "Host: `tr -d '()' <<< $host`"
        echo "Groups`groups $user | grep -o ':.*$'`"
        echo "ID: `id $user`"
        echo
done <<< "$whoout"
```

Rather than storing the output of who in a shell variable, we could call it here with back quotes.

Here String (<<<)

Example 6:

/cs2211/week6/ex6b.sh

```
#!/bin/bash

while read user pts login time host; do
        echo "$user:"
        echo "PTS: $pts"
        echo "Host: `tr -d '()' <<< $host`"
        echo "Groups`groups $user | grep -o ':.*$'`"
        echo "ID: `id $user`"
        echo
done <<< "`who`"
```

Rather than storing the output of who in a shell variable, we could call it here with back quotes.

Optional Readings

Will not be tested on the following:

- The conditional expression ([[ ]]):
  http://wiki.bash-hackers.org/syntax/ccmd/conditional_expression

- Calculating with dc:
  http://wiki.bash-hackers.org/howto/calculate-dc

- `trap` (Chapter 13.19)
- Shell Functions (Chapter 13.18)
- `eval` (Chapter 13.20)
- `exec` (Chapter 13.21)

Chapter 13
Practice Problems

CH13 Practice Problems

13.1 When the script `foo -l -t bar[1-3]` runs, what values do $# and $* acquire?

CH13 Practice Problems

13.2 Use a script to take two numbers as arguments and output their sum using bc and expr. Include error checking to test that two arguments were entered.

CH13 Practice Problems

13.3 If x has the value 5, and you reassign it with x="expr $x + 10", what is the new value of x? What would have been the value if single quotes had been used?

CH13 Practice Problems

13.9 You have to run a job at night and need to have both the output and error messages in the same file. How do you run the script?

CH13 Practice Problems

13.10 Write a script that behaves in both interactive and noninteractive mode. When no arguments are supplied, it picks up each C program from the current directory and list the first 10 lines. It then prompts for deletion of the file. If the user supplies and arguments with the script, it works on those files only.

CH13 Practice Problems

13.18 Write a script that accepts a 10-digit number as an argument and writes it to the standard output in the form nnn-nnn-nnnn. Perform validation checks to ensure that:

1) **A single argument is entered.**
2) **The number can't begin with 0.**
3) **The number comprises 10 digits.**