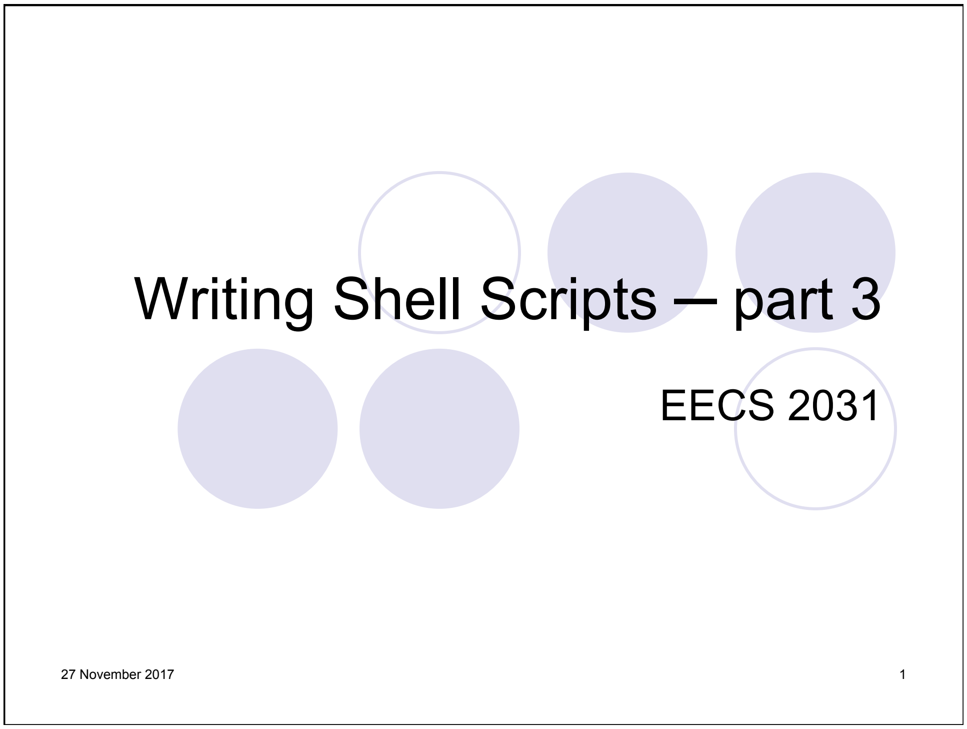
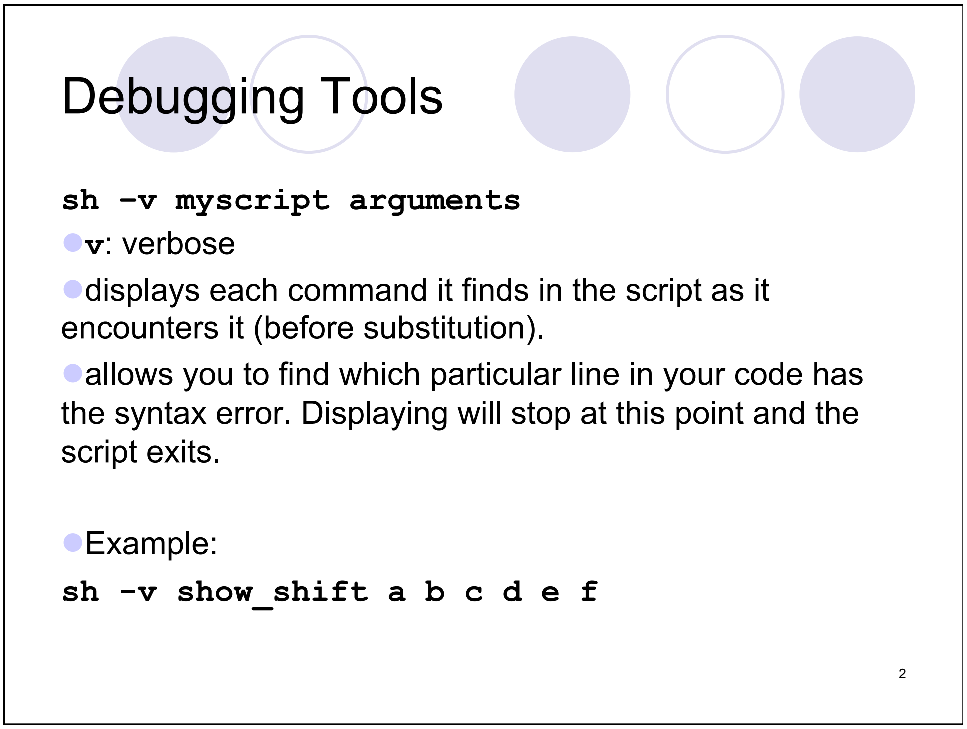


Writing Shell Scripts – part 3

EECS 2031

27 November 2017



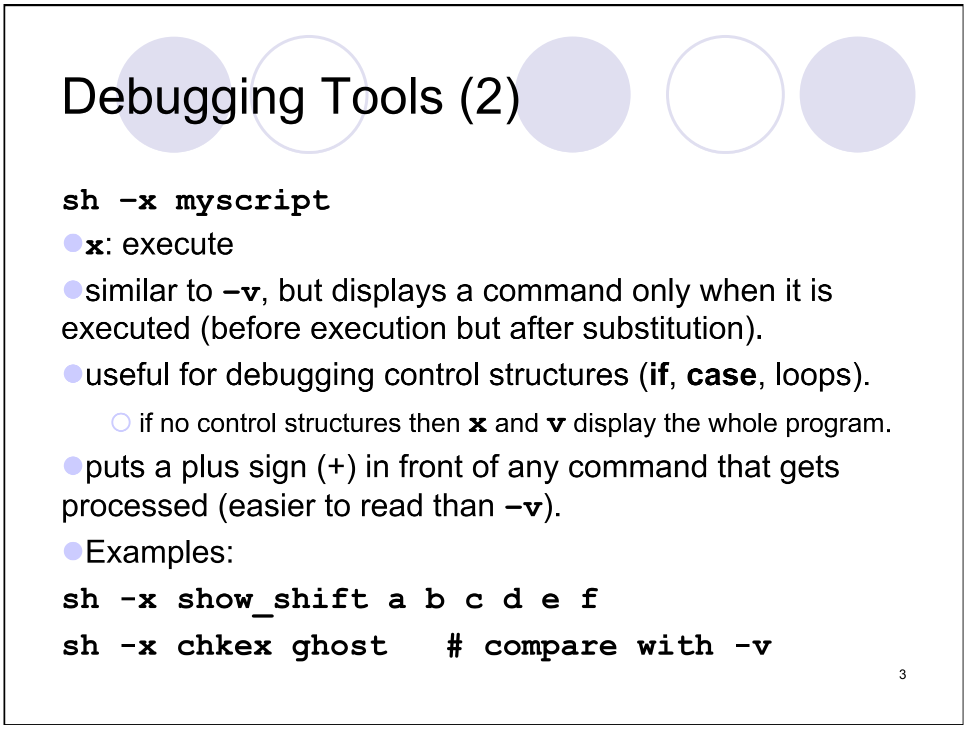
Debugging Tools

```
sh –v myscript arguments
```

- **v**: verbose
- displays each command it finds in the script as it encounters it (before substitution).
- allows you to find which particular line in your code has the syntax error. Displaying will stop at this point and the script exits.

- Example:

```
sh -v show_shift a b c d e f
```

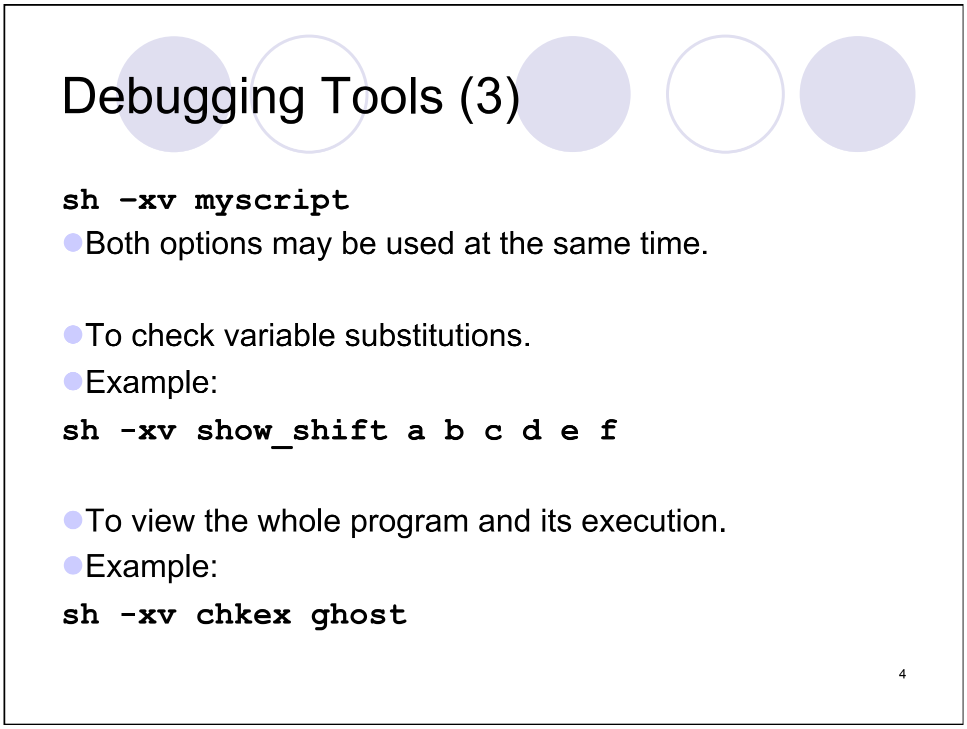


Debugging Tools (2)

```
sh –x myscript
```

- **x**: execute
- similar to **–v**, but displays a command only when it is executed (before execution but after substitution).
- useful for debugging control structures (**if**, **case**, loops).
  - if no control structures then **x** and **v** display the whole program.
- puts a plus sign (+) in front of any command that gets processed (easier to read than **–v**).
- Examples:

```
sh -x show_shift a b c d e f
sh -x chkex ghost   # compare with -v
```



Debugging Tools (3)

```
sh –xv myscript
```

- Both options may be used at the same time.

- To check variable substitutions.
- Example:

```
sh -xv show_shift a b c d e f
```

- To view the whole program and its execution.
- Example:

```
sh -xv chkex ghost
```

Debugging Tools (4)

`sh -n myscript`

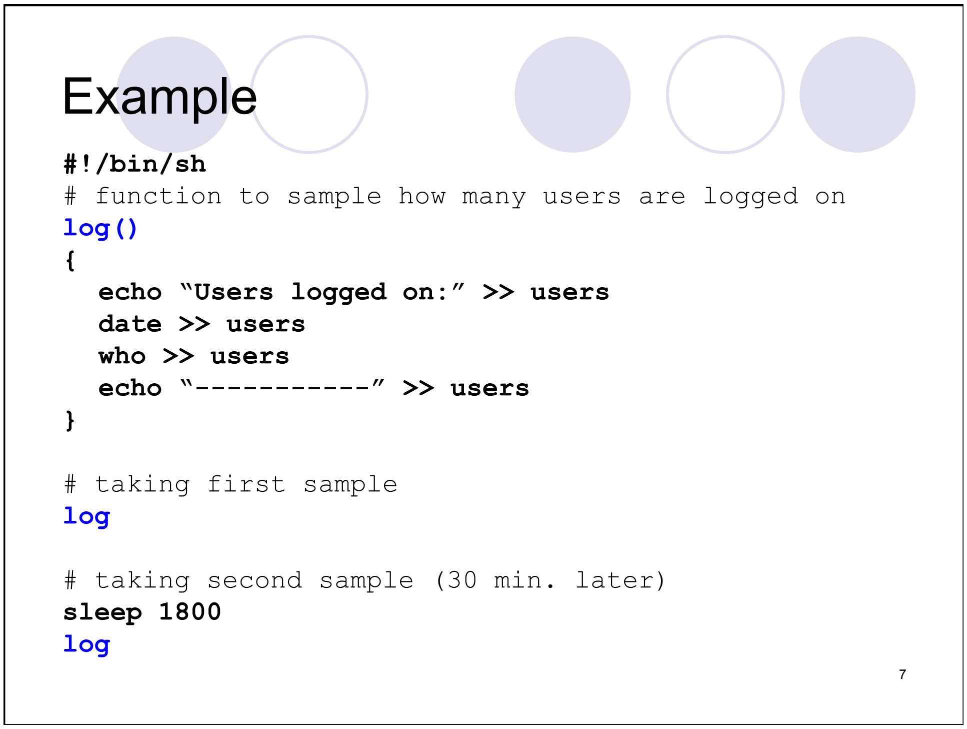
- Reads the commands but does NOT execute them.
- Useful for “compiling” the script to detect syntax errors.
- Example uses:
  - a good working script will modify/delete files.
  - interactive input from user is required.
  - very long scripts.

Shell Functions

- Similar to shell scripts.
- Stored in shell where it is defined (instead of in a file).
- Executed within **sh**
  - no child process spawned
- Syntax:

```
function_name()
{
  commands
}
```

- Allows structured shell scripts



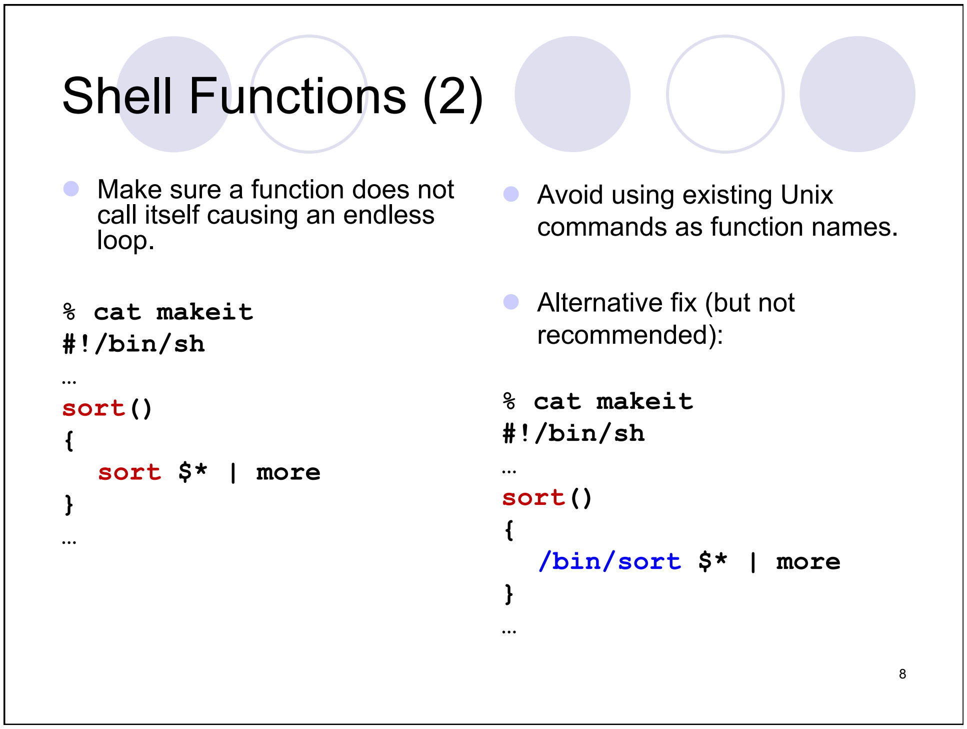
Example

```
#!/bin/sh
# function to sample how many users are logged on
log()
{
  echo “Users logged on:” >> users
  date >> users
  who >> users
  echo “-----------” >> users
}

# taking first sample
log

# taking second sample (30 min. later)
sleep 1800
log
```

7



Shell Functions (2)

- Make sure a function does not call itself causing an endless loop.

```
% cat makeit
#!/bin/sh
…
sort()
{
  sort $* | more
}
…
```

- Avoid using existing Unix commands as function names.
- Alternative fix (but not recommended):

```
% cat makeit
#!/bin/sh
…
sort()
{
  /bin/sort $* | more
}
…
```

8

Changing Values of Positional Parameters

- Positional parameters **`$1, $2,`** … normally store command line arguments.
- Their values can be changed using **`set`** command , for example, **`set `date``**
- The new values are the output of **`date`** command.

Example

```
% cat setparm
#!/bin/sh
echo "Hello, $1. You entered $# command line argument(s). Today's date is ..."
date
set `date`
echo There are now $# positional parameters.  The new parameters are ...
echo \$1 = $1, \$2 = $2, \$3 = $3, \$4 = $4, \$5 = $5, \$6 = $6.

% setparm Amy Tony
Hello, Amy. You entered 2 command line argument(s). Today's date is ...
Sat Nov 27 11:55:52 EST 2010
There are now 6 positional parameters. The new parameters are ...
$1 = Sat, $2 = Nov, $3 = 27, $4 = 11:55:52, $5 = EST, $6 = 2010.
```

Environment and Shell Variables

- Standard UNIX variables are divided into 2 categories: shell variables and environment variables.
- **Shell variables**: apply only to the current instance of the shell; used to set short-term working conditions.
  - displayed using `'set'` command.
- **Environment variables**: set at login and are valid for the duration of the session.
  - displayed using `'env'` command.
- By convention,
  - shell variables have lower case names.
  - environment variables have UPPER case names

Environment and Shell Variables (2)

- In general, environment and shell variables that have “the same” name (apart from the case) are distinct and independent, except for possibly having the same initial values.

Exceptions:

- When **`home`**, **`user`** and **`term`** are changed, **`HOME`**, **`USER`** and **`TERM`** receive the same values.
- But changing **`HOME`**, **`USER`** or **`TERM`** does not affect **`home`**, **`user`** or **`term`**.
- Changing **`PATH`** causes **`path`** to be changed **and vice versa**.

Variable path

- PATH and path specify directories to search for commands and programs.

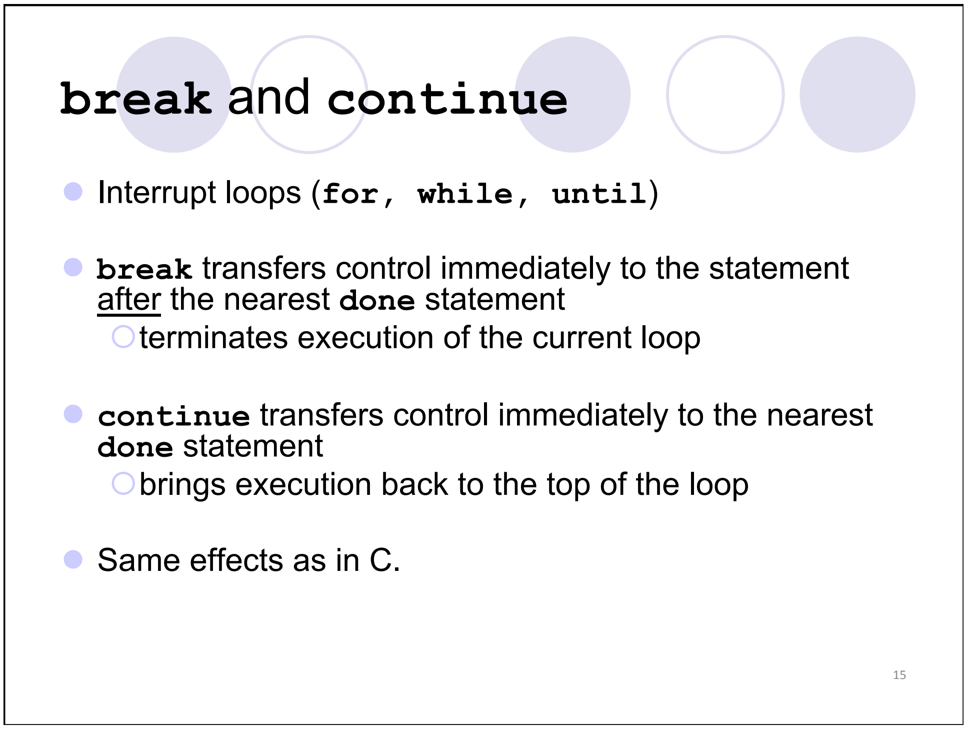
```
cd                # move to home directory
countargs a b     # in C2031/Lect_UNIX, so failed
echo $path
set path=($path C2031/Lect_UNIX)
echo $path
countargs a b     # successful
```

- To add a path permanently, add the line to your .cshrc file after the list of other commands.

```
set path=($path .)     # to avoid typing ./a.out
```

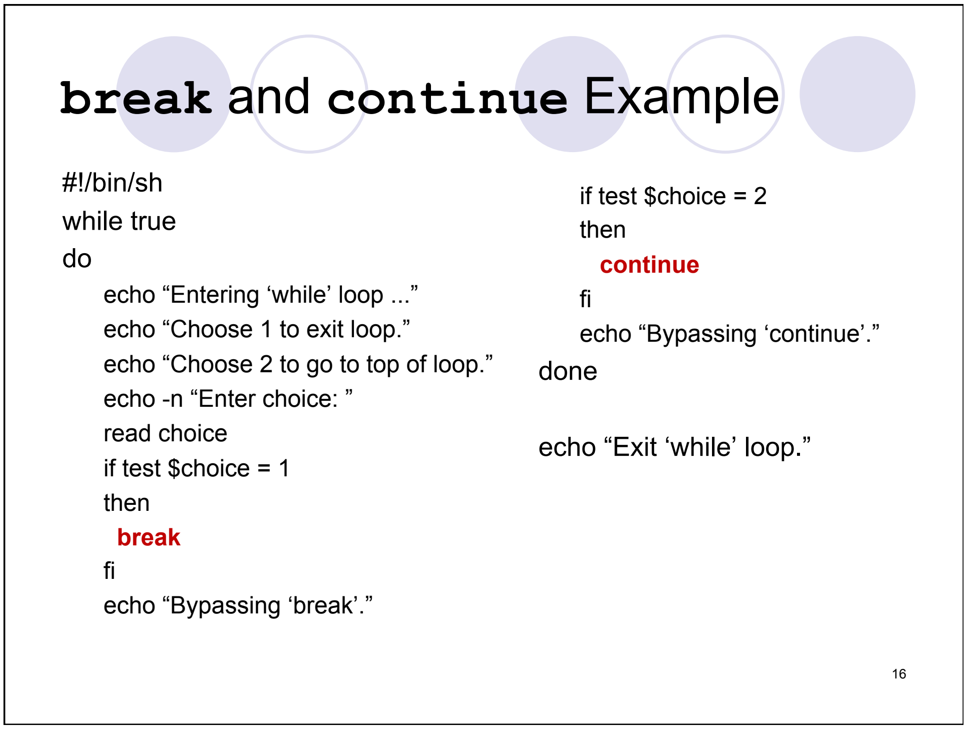
set Command Summary

- Displays shell variables
- Set variables (e.g., set path)
- Changing command-line arguments
  - `set `date``



break and **continue**

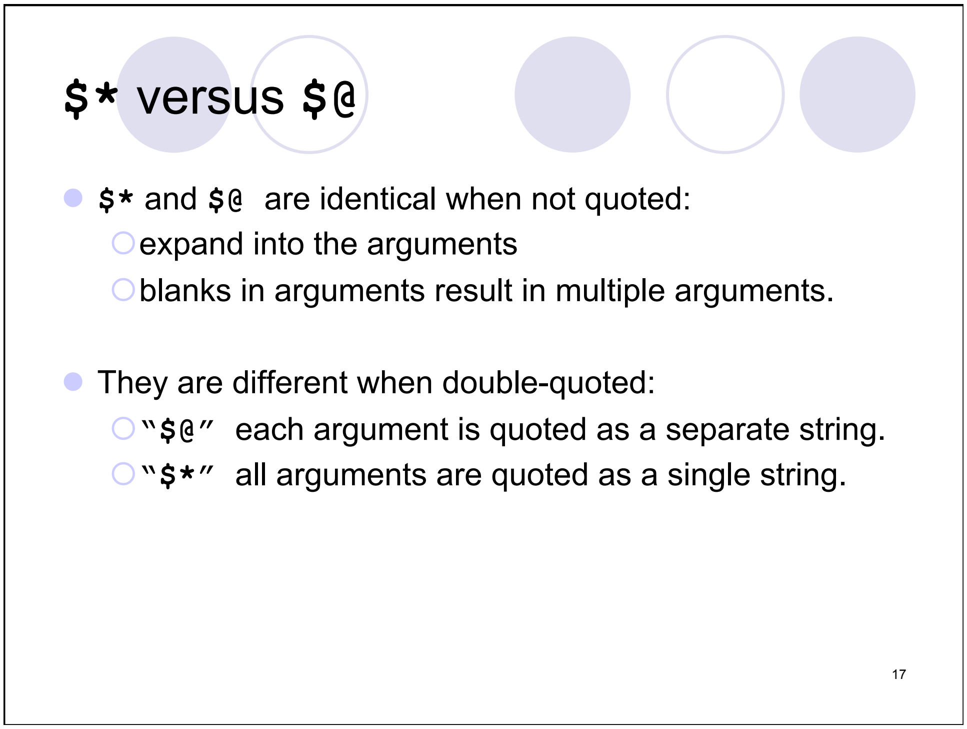
- Interrupt loops (**for, while, until**)
- **break** transfers control immediately to the statement after the nearest **done** statement
  - terminates execution of the current loop
- **continue** transfers control immediately to the nearest **done** statement
  - brings execution back to the top of the loop
- Same effects as in C.



break and **continue** Example

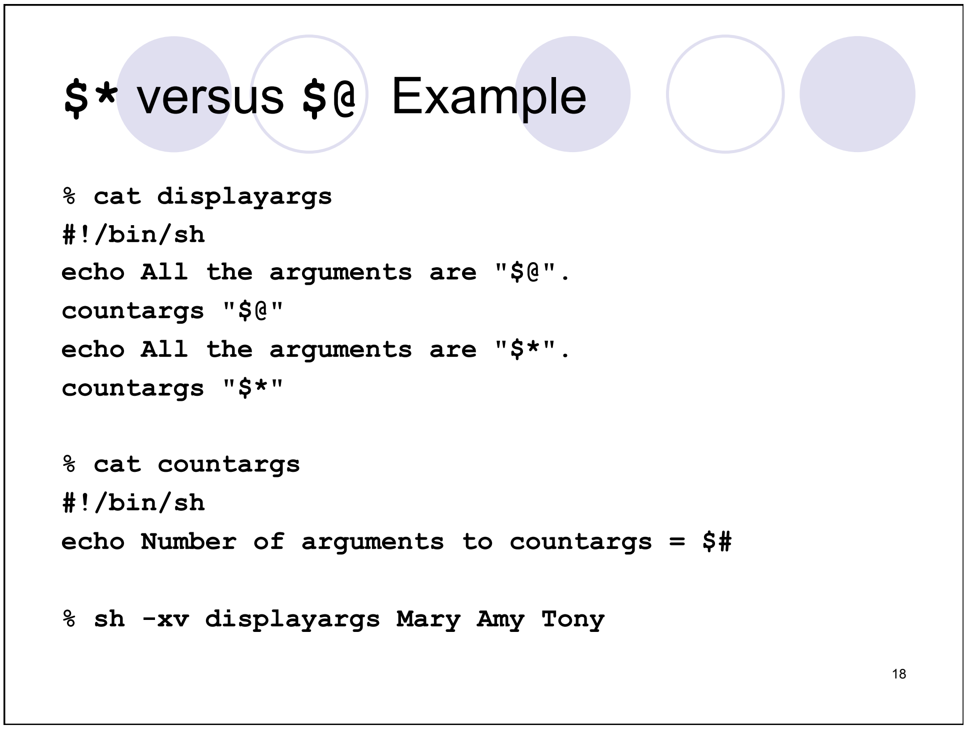
```
#!/bin/sh
while true
do
    echo “Entering ‘while’ loop ...”
    echo “Choose 1 to exit loop.”
    echo “Choose 2 to go to top of loop.”
    echo -n “Enter choice: ”
    read choice
    if test $choice = 1
    then
      break
    fi
    echo “Bypassing ‘break’.”
    if test $choice = 2
    then
      continue
    fi
    echo “Bypassing ‘continue’.”
done

echo “Exit ‘while’ loop.”
```



`$*` versus `$@`

- `$*` and `$@` are identical when not quoted:
  - expand into the arguments
  - blanks in arguments result in multiple arguments.
- They are different when double-quoted:
  - `“$@”` each argument is quoted as a separate string.
  - `“$*”` all arguments are quoted as a single string.

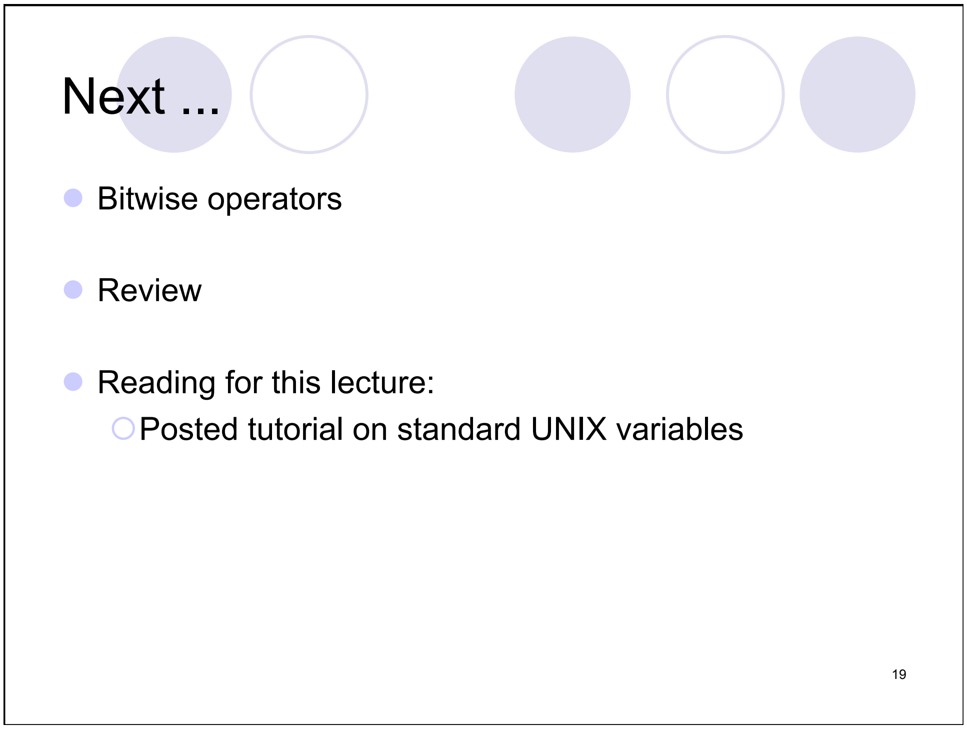


`$*` versus `$@` Example

```
% cat displayargs
#!/bin/sh
echo All the arguments are "$@".
countargs "$@"
echo All the arguments are "$*".
countargs "$*"

% cat countargs
#!/bin/sh
echo Number of arguments to countargs = $#

% sh -xv displayargs Mary Amy Tony
```



Next ...

- Bitwise operators
- Review
- Reading for this lecture:
  - Posted tutorial on standard UNIX variables