UNIX Shell Scripts

CSE 2031

Fall 2012

What Is a Shell?

- A program that interprets your requests to run other programs
- Most common Unix shells:
  - Bourne shell (sh)
  - C shell (csh - tcsh)
  - Korn shell (ksh)
  - Bourne-again shell (bash)
- In this course we focus on Bourne shell (sh).

The Bourne Shell

- A high level programming language
- Processes groups of commands stored in files called *scripts*
- Includes
  - variables
  - control structures
  - processes
  - signals

Executable Files

- Contain one or more shell commands.
- These files can be made *executable.*
- # indicates a comment
  - Except on line 1 when followed by an "!"

```
% cat welcome
#!/bin/sh
echo 'Hello World!'
```

Executable Files: Example

```
% cat welcome
#!/bin/sh
echo 'Hello World!'
% welcome
welcome: execute permission denied
% chmod 755 welcome
% ls -l welcome
-rwxr-xr-x 1 bil faculty 30 Nov 12 10:49 welcome
% welcome
Hello World!
% welcome > greet_them
% cat greet_them
Hello World!
```

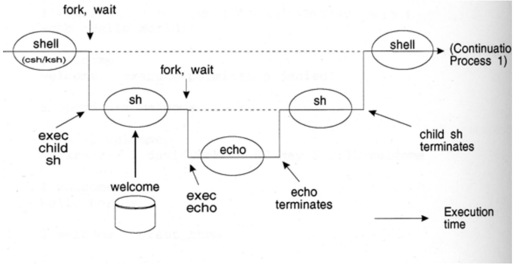
Executable Files (cont.)

- If the file is not executable, use "sh" followed by the file name to run the script.
- Example:

```
% chmod 644 welcome
% ls -l welcome
-rw-r--r-- 1 bil faculty 30 Nov 12 10:49 welcome
% sh welcome
Hello World!
```

Processes

Consider the welcome program.



Processes: Explanation

- Every program is a “child” of some other program.
- Shell fires up a child shell to execute script.
- Child shell fires up a new (grand)child process for each command.
- Shell (parent) sleeps while child executes.
- Every process (executing a program) has a unique PID.
- Parent does not sleep while running background processes.

Process-Related Variables

- Variable `$$` is PID of the shell.

```
% cat shpid
#!/bin/sh
ps
echo PID of shell is = $$

% shpid
 PID  TTY          TIME CMD
 5658 pts/75   00:00:00 shpid
 5659 pts/75   00:00:00 ps
11231 pts/75   00:00:00 tcsh
PID of shell is = 5658
```

Process Exit Status

- All processes return exit status (return code).
- Exit status tells us whether the last command was successful or not.
- Stored in variable `$?`
- 0 (zero) means command executed successfully.
- 0 is good; non-zero is bad.
- Good practice: Specify your own exit status in a shell script using `exit` command.
  - ○default value is 0 (if no exit code is given).

Process Exit Status: Example

- A more talkative grep.

```
% cat igrep
#!/bin/sh
# Arg 1: search pattern
# Arg 2: file to search
#
grep $1 $2
if test  $? -ne 0
then
  echo Pattern not found.
fi
```

```
% igrep echo phone
echo -n “Enter name: ”

% igrep echo2 chex
Pattern not found.
```

Redirection tricks

- Want to run a command to check its exit status and ignore the output?
  `diff f1 f2 > /dev/null`
- Want to combine standard error and standard output?
  `diff f1 f2 > /dev/null 2>&1`

Variables: Three Types

- Standard UNIX variables
  - Consist of shell variables and environment variables.
  - Used to tailor the operating environment to suit your needs.
  - Examples: TERM, HOME, PATH
  - To display your environment variables, type “set”.
- User variables: variables you create yourself.
- Positional parameters
  - Also called read-only variables, automatic variables.
  - Store the values of command-line arguments.

User Variables

- Syntax: **`name=value`**
- **No space around the equal sign!**
- **All shell variables store strings** (no numeric values).
- Variable name: combinations of letters, numbers, and underscore character (_) that do not start with a number.
- Avoid existing commands and environment variables.
- Shell stores and remembers these variables and supplies value on demand.

User Variables

- To use a variable: **`$varname`**
- Operator **`$`** tells the shell to substitute the value of the variable name.

```
% cat ma
#!/bin/sh
dir=/usr/include/
echo $dir
echo dir
ls $dir | grep 'ma’
```

echo and variables

- What if I want to display the following?

`$dir`

- Two ways to prevent variable substitution:

```
echo ‘$dir’
echo \$dir
```

- Note:

`echo “$dir”` does the same as

`echo $dir`

User Variables and Quotes

- If **`value`** contains no space, no need to use quotes: **`dir=/usr/include/`**
- Unless you want to protect the literal **`$`**

```
% cat quotes
#!/bin/sh
# Test values with quotes
myvar1=$100
myvar2='$100'
echo The price is $myvar1
echo The price is $myvar2
```

User Variables and Quotes

- If **`value`** contains one or more spaces:
- Use single quotes for NO interpretation of metacharacters (protect the literal)
- Use double quotes for interpretation of metacharacters

Example

```
% cat quotes2
#!/bin/sh
myvar=`whoami`
squotes='Today is `date`, $myvar.'
dquotes="Today is `date`, $myvar."
echo $squotes
echo $dquotes
```

Example

```
% cat twodirs
#!/bin/sh
# The following needs quotes
dirs="/usr/include/ /usr/local/"
echo $dirs
ls -l $dirs
```

Command Line Arguments

- Command line arguments stored in variables are called positional parameters.
- These parameters are named **$1** through **$9**.
- Command itself is in parameter **$0**.
- In diagram format:

```
command arg1 arg2 arg3 arg4 arg5 arg6 arg7 arg8 arg9
   $0     $1   $2   $3   $4   $5   $6   $7   $8   $9
```

Example 1

```
% cat showargs
#!/bin/sh
echo First four arguments from the
echo command line are: $1 $2 $3 $4

% showargs William Mary Richard James
First four arguments from the
command line are: William Mary Richard James
```

Example 2

```
% cat chex
#!/bin/sh
# Make a file executable
chmod u+x $1
echo $1 is now executable:
ls –l $1

% sh chex chex
chex is now executable:
-rwx------  1 bil faculty 86 Nov 12 11:34 chex

% chex showargs
showargs is now executable:
-rwx------  1 bil faculty 106 Nov  2 14:26 showargs
```

Command Line Arguments

`$#` represents the number of command line arguments
`$*` represents all the command line arguments
`$@` represents all the command line arguments

```
% cat check_args
#!/bin/sh
echo “There are $# arguments.”
echo “All the arguments are: $*”
# or echo “All the arguments are: $@”

% check_args Mary Tom Amy Tony
There are 4 arguments.
All the arguments are: Mary Tom Amy Tony
```

Command Line Arguments

- `$#` does NOT include the program name (unlike argc in C programs)
- `$*` 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 = $#

% displayargs Mary Amy Tony
```

Control Structures

- if then else
- for
- while
- case (which)
- until

if Statement and test Command

- Syntax:

```
if condition
then
   command(s)
elif condition_2
then
   command(s)
else
   command(s)
fi
```

- Command `test` is often used in *`condition`*.

if – then – else Example

```
% cat if_else
#!/bin/sh
echo -n 'Enter string 1: '
read string1
echo -n 'Enter string 2: '
read string2
if test $string1 = $string2
then
        echo 'They match!'
else
        echo 'No match!'
fi
```

```
% if_else
Enter string 1: acd
Enter string 2: 123
No match!

% if_else
Enter string 1: 123
Enter string 2: 123
They match!
```

`test` Command

-e arg	True if arg exists
-d arg	True if arg is a directory
-f arg	True if arg is an ordinary file
-r arg	True if arg is readable
-w arg	True if arg is writable
-x arg	True if arg is executable
-s arg	True if size of arg is greater than 0
! –d arg	True if arg is not a directory

`test` Command (Numeric tests)

n1 –eq n2	n1 == n2
n1 –ge n2	n1 >= n2
n1 –gt n2	n1 > n2
n1 –le n2	n1 <= n2
n1 –ne n2	n1 != n2
n1 –lt n2	n1 < n2

Parentheses can be used to group conditions.

`test` Example 1

```
% cat check_file
if test ! -e $1
then
   echo "$1 does not exist."
   exit 1
else
   ls -l $1
fi
```

`test` Example 2

```
% cat check_file2
#!/bin/sh
if test $# -eq 0
then
   echo Usage: check_file file_name
   exit 1
fi
…
```

`test` Example 3

- What is wrong with the following script?

```
% cat chkex2
#!/bin/sh
# Check if a file is executable.
if test -x $1
then
   echo File $1 is executable.
else
   echo File $1 is not executable.
fi
```

`test` and Logical Operators

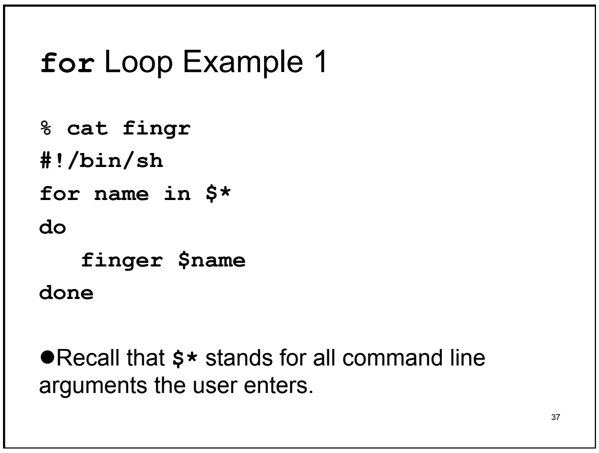
- `!`, `||` and `&&` as in C
- Following is better version of `test` Example 3

```
%cat chkex
#!/bin/sh
if test -e $1 && test -x $1
then
   echo File $1 is executable.
elif test ! -e $1
then
   echo File $1 does not exist.
else
   echo File $1 is not executable.
fi
```

`for` Loops

```
for variable in list
do
   command(s)
done
```

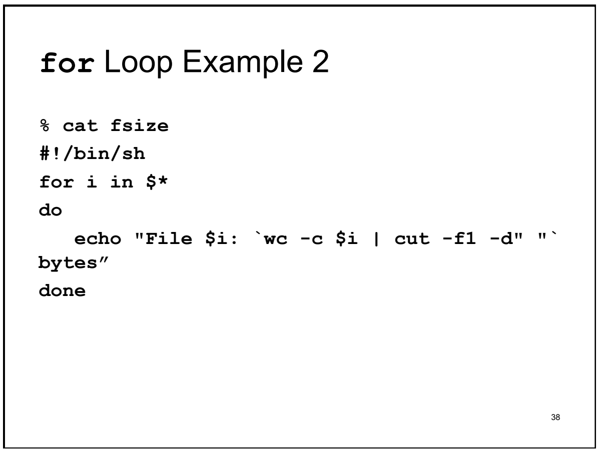
- *`variable`* is a user-defined variable.
- *`list`* is a sequence of strings separated by spaces.



`for` Loop Example 1

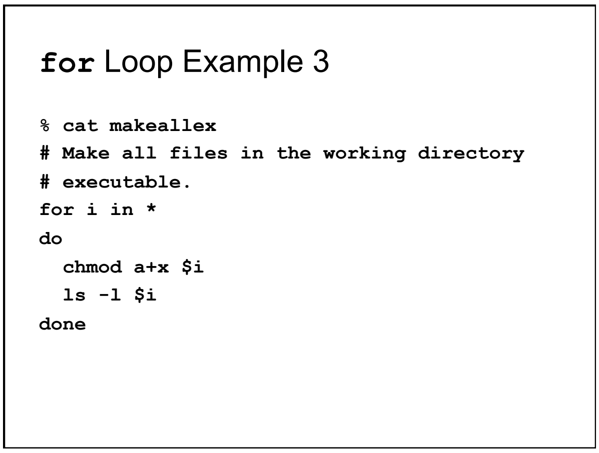
```
% cat fingr
#!/bin/sh
for name in $*
do
   finger $name
done
```

- Recall that `$*` stands for all command line arguments the user enters.



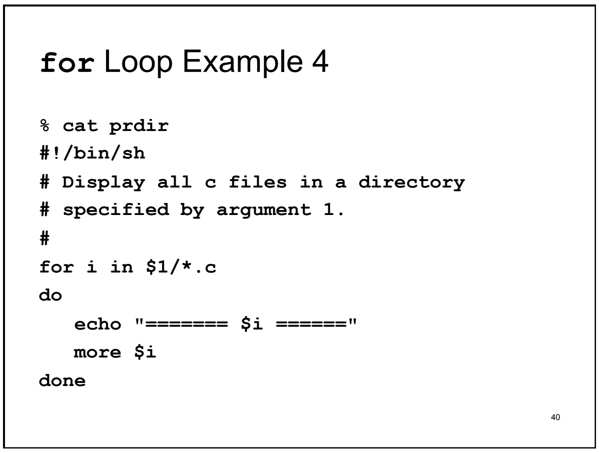
`for` Loop Example 2

```
% cat fsize
#!/bin/sh
for i in $*
do
   echo "File $i: `wc -c $i | cut -f1 -d" "` bytes″
done
```



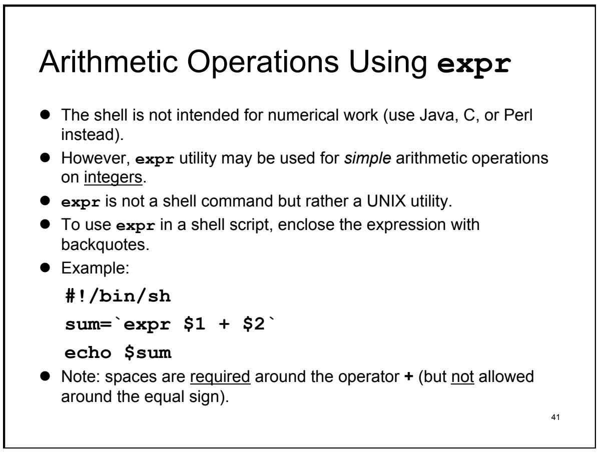
`for` Loop Example 3

```
% cat makeallex
# Make all files in the working directory
# executable.
for i in *
do
  chmod a+x $i
  ls -l $i
done
```



`for` Loop Example 4

```
% cat prdir
#!/bin/sh
# Display all c files in a directory
# specified by argument 1.
#
for i in $1/*.c
do
   echo "======= $i ======"
   more $i
done
```

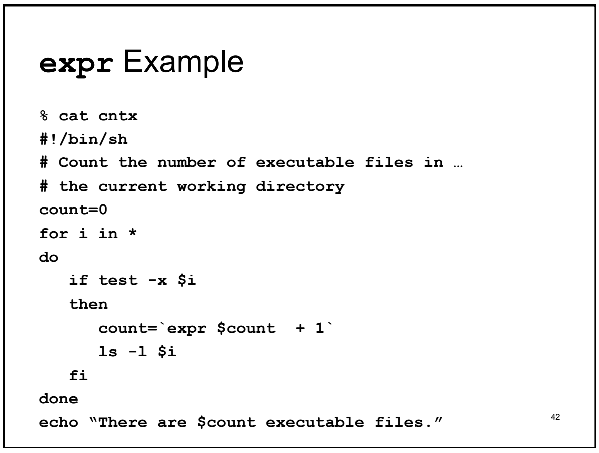


Arithmetic Operations Using `expr`

- The shell is not intended for numerical work (use Java, C, or Perl instead).
- However, `expr` utility may be used for *simple* arithmetic operations on integers.
- `expr` is not a shell command but rather a UNIX utility.
- To use `expr` in a shell script, enclose the expression with backquotes.
- Example:

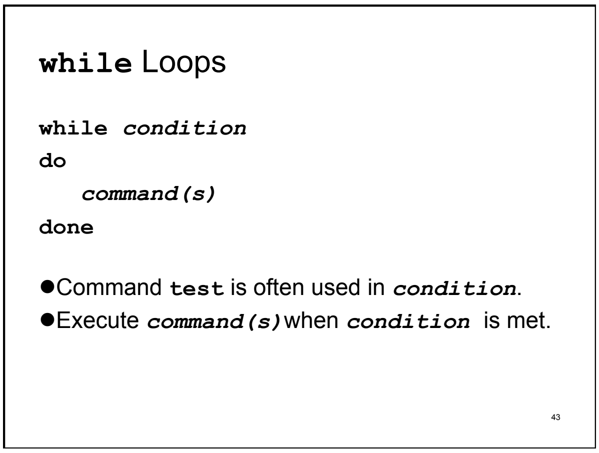
```
#!/bin/sh
sum=`expr $1 + $2`
echo $sum
```

- Note: spaces are required around the operator **+** (but not allowed around the equal sign).



`expr` Example

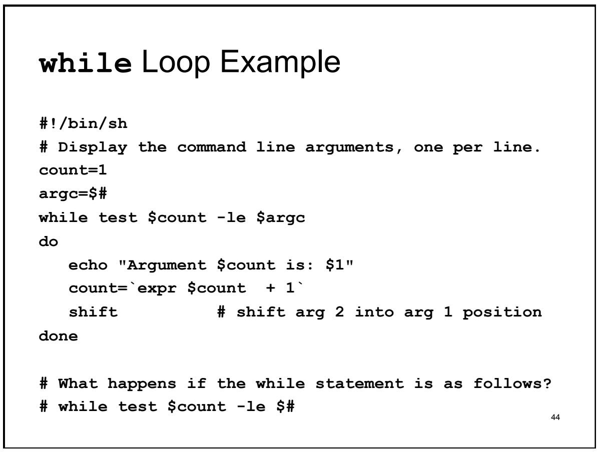
```
% cat cntx
#!/bin/sh
# Count the number of executable files in …
# the current working directory
count=0
for i in *
do
   if test -x $i
   then
      count=`expr $count  + 1`
      ls -l $i
   fi
done
echo “There are $count executable files.”
```



while Loops

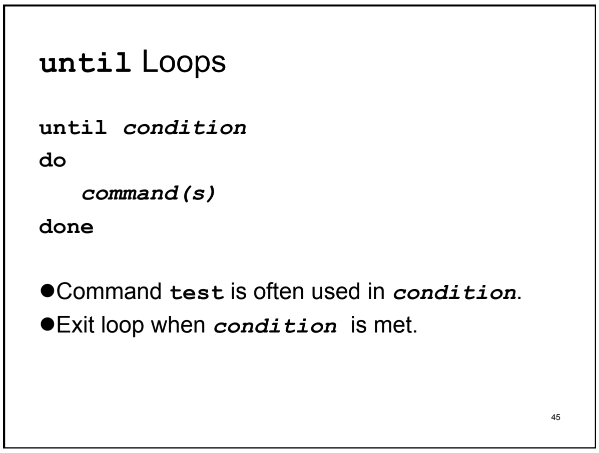
```
while condition
do
   command(s)
done
```

- Command **test** is often used in ***condition***.
- Execute ***command(s)*** when ***condition*** is met.



while Loop Example

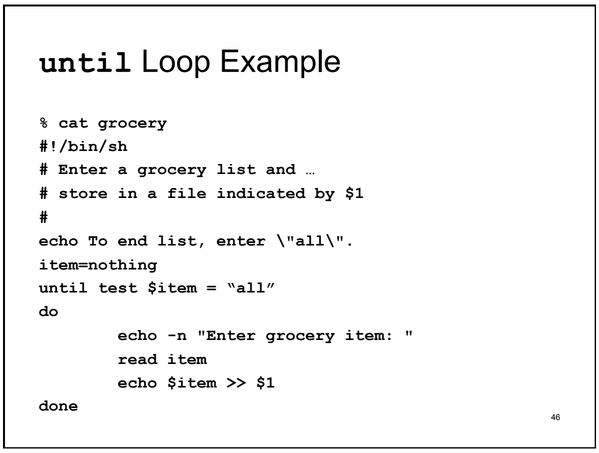
```
#!/bin/sh
# Display the command line arguments, one per line.
count=1
argc=$#
while test $count -le $argc
do
   echo "Argument $count is: $1"
   count=`expr $count  + 1`
   shift          # shift arg 2 into arg 1 position
done

# What happens if the while statement is as follows?
# while test $count -le $#
```



until Loops

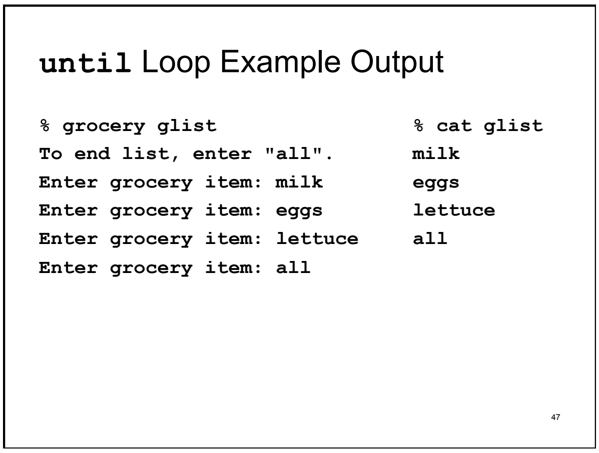
```
until condition
do
   command(s)
done
```

- Command **test** is often used in ***condition***.
- Exit loop when ***condition*** is met.



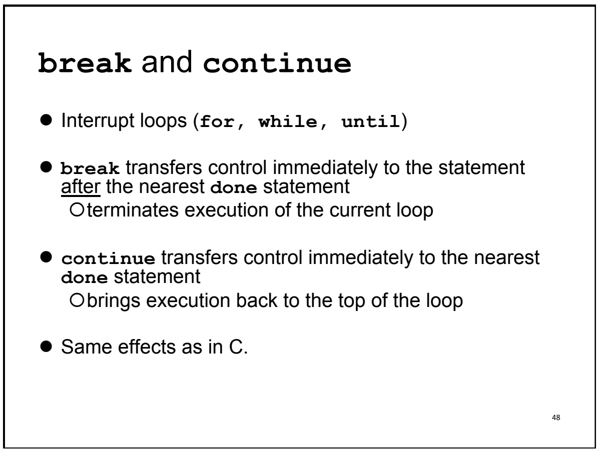
until Loop Example

```
% cat grocery
#!/bin/sh
# Enter a grocery list and …
# store in a file indicated by $1
#
echo To end list, enter \"all\".
item=nothing
until test $item = “all”
do
        echo -n "Enter grocery item: "
        read item
        echo $item >> $1
done
```



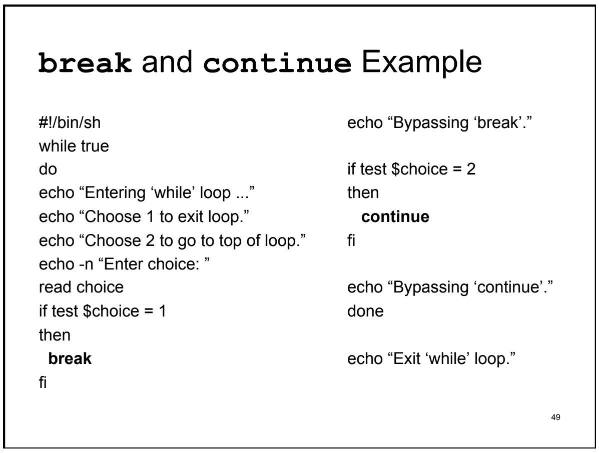
until Loop Example Output

```
% grocery glist               % cat glist
To end list, enter "all".     milk
Enter grocery item: milk      eggs
Enter grocery item: eggs      lettuce
Enter grocery item: lettuce   all
Enter grocery item: all
```



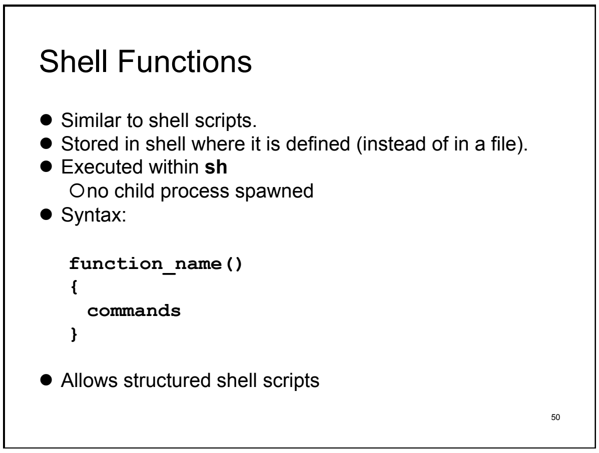
break and **continue**

- 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.”
```

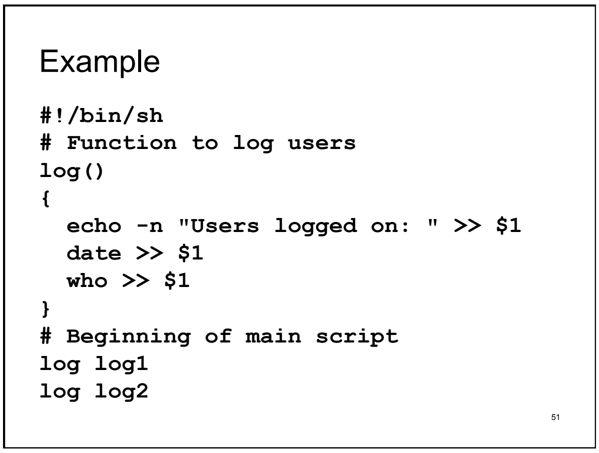


Shell Functions

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

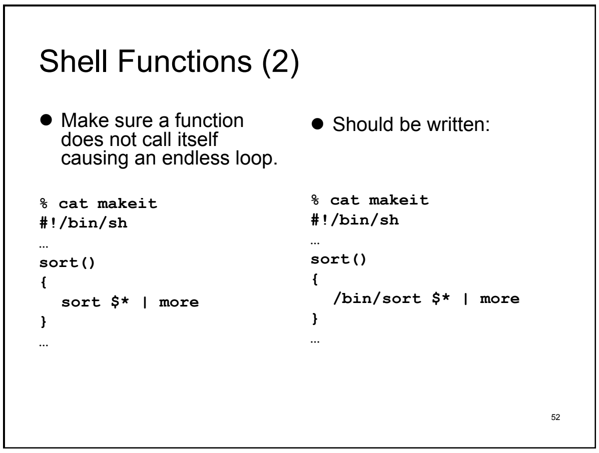
```
function_name()
{
  commands
}
```

- Allows structured shell scripts



Example

```
#!/bin/sh
# Function to log users
log()
{
  echo -n "Users logged on: " >> $1
  date >> $1
  who >> $1
}
# Beginning of main script
log log1
log log2
```



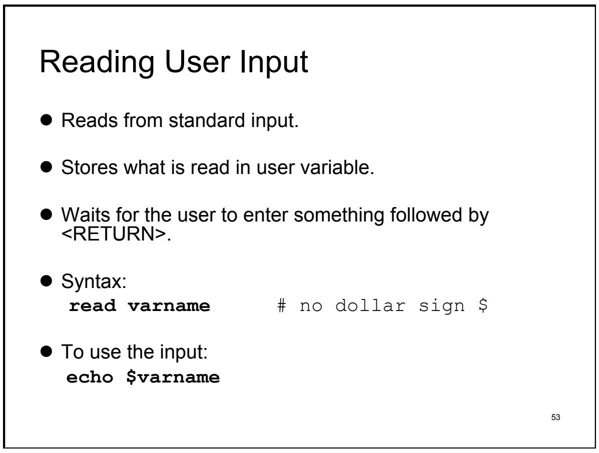
Shell Functions (2)

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

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

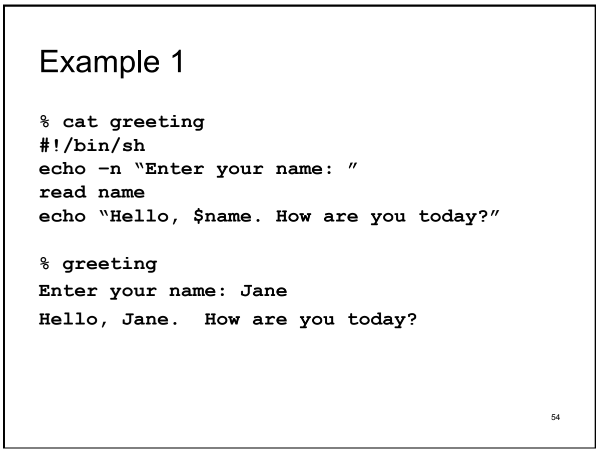
- Should be written:

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



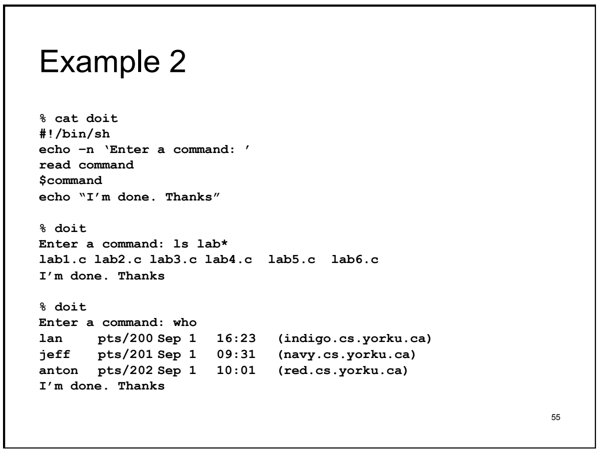
Reading User Input

- Reads from standard input.
- Stores what is read in user variable.
- Waits for the user to enter something followed by <RETURN>.
- Syntax:
  ```
  read varname      # no dollar sign $
  ```
- To use the input:
  ```
  echo $varname
  ```



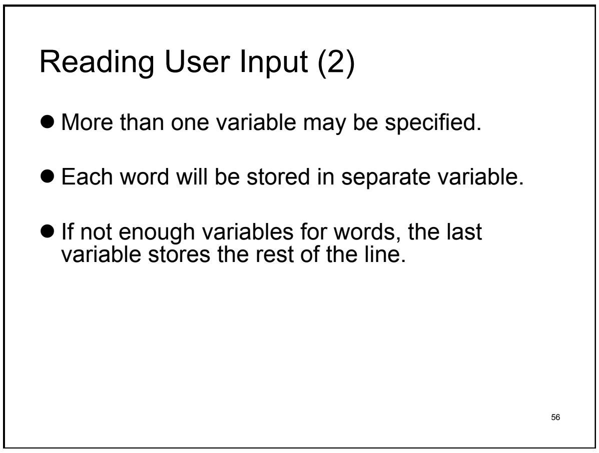
Example 1

```
% cat greeting
#!/bin/sh
echo –n “Enter your name: ”
read name
echo “Hello, $name. How are you today?”

% greeting
Enter your name: Jane
Hello, Jane.  How are you today?
```



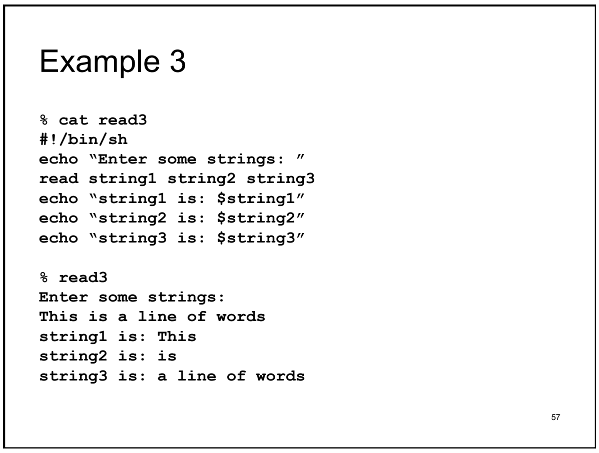
Example 2

```
% cat doit
#!/bin/sh
echo –n ‘Enter a command: ’
read command
$command
echo “I’m done. Thanks”

% doit
Enter a command: ls lab*
lab1.c lab2.c lab3.c lab4.c  lab5.c  lab6.c
I’m done. Thanks

% doit
Enter a command: who
lan     pts/200 Sep 1   16:23   (indigo.cs.yorku.ca)
jeff    pts/201 Sep 1   09:31   (navy.cs.yorku.ca)
anton   pts/202 Sep 1   10:01   (red.cs.yorku.ca)
I’m done. Thanks
```



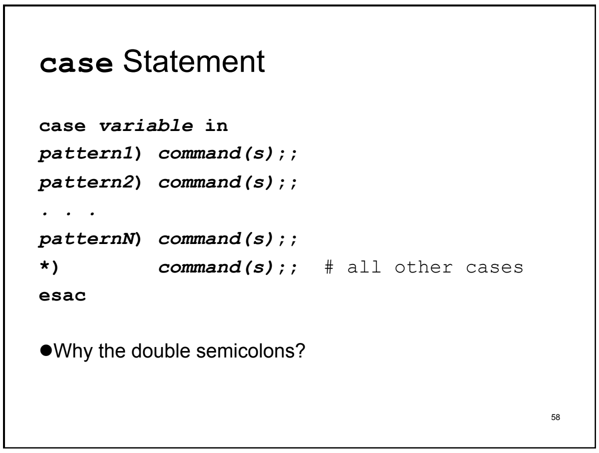
Reading User Input (2)

- More than one variable may be specified.
- Each word will be stored in separate variable.
- If not enough variables for words, the last variable stores the rest of the line.



Example 3

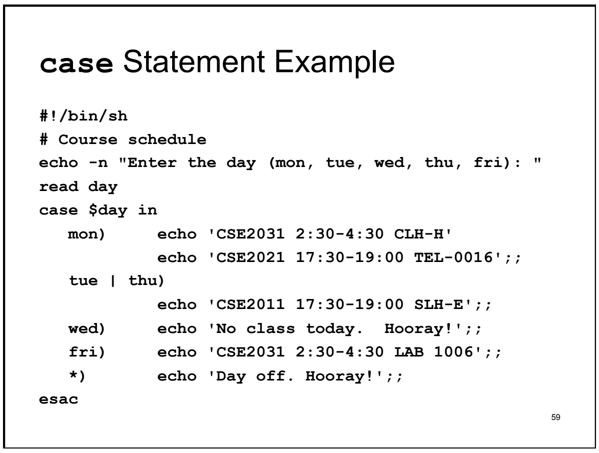
```
% cat read3
#!/bin/sh
echo “Enter some strings: ”
read string1 string2 string3
echo “string1 is: $string1”
echo “string2 is: $string2”
echo “string3 is: $string3”

% read3
Enter some strings:
This is a line of words
string1 is: This
string2 is: is
string3 is: a line of words
```



case Statement

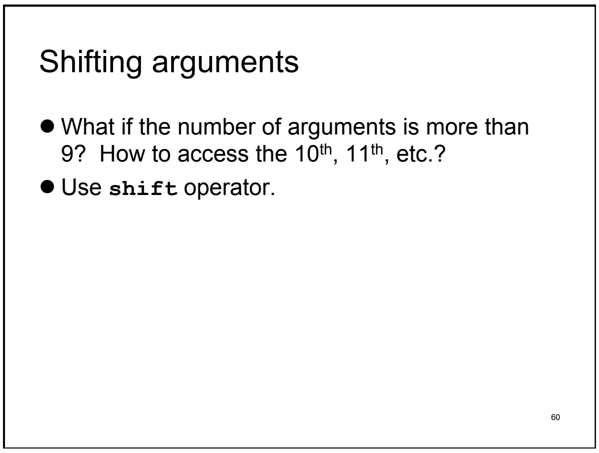
```
case variable in
pattern1) command(s);;
pattern2) command(s);;
. . .
patternN) command(s);;
*)        command(s);;  # all other cases
esac
```

- Why the double semicolons?



case Statement Example

```
#!/bin/sh
# Course schedule
echo -n "Enter the day (mon, tue, wed, thu, fri): "
read day
case $day in
   mon)     echo 'CSE2031 2:30-4:30 CLH-H'
            echo 'CSE2021 17:30-19:00 TEL-0016';;
   tue | thu)
            echo 'CSE2011 17:30-19:00 SLH-E';;
   wed)     echo 'No class today.  Hooray!';;
   fri)     echo 'CSE2031 2:30-4:30 LAB 1006';;
   *)       echo 'Day off. Hooray!';;
esac
```



Shifting arguments

- What if the number of arguments is more than 9? How to access the 10th, 11th, etc.?
- Use `shift` operator.

shift Operator

- **shift** promotes each argument one position to the left.
- Allows access to arguments beyond $9.
- Operates as a conveyor belt.
  Shifts contents of $2 into $1
  Shifts contents of $3 into $2
  Shifts contents of $4 into $3 etc.
- Eliminates argument that used to be in $1
- After a shift, the argument count stored in `$#` is automatically decreased by one.

Example 1

```
% cat shiftex
#!/bin/sh
echo "arg1 = $1, arg8 = $8, arg9 = $9, ARGC = $#"
myvar=$1    # save the first argument
shift
echo "arg1 = $1, arg8 = $8, arg9 = $9, ARGC = $#"
echo "myvar = $myvar"

% shiftex 1 2 3 4 5 6 7 8 9 10 11 12
arg1 = 1, arg8 = 8, arg9 = 9, ARGC = 11
arg1 = 2, arg8 = 9, arg9 = 10, ARGC = 10
myvar = 1
```

Example 2

```
% cat show_shift
#!/bin/sh
echo "arg1=$1, arg2=$2, arg3=$3"
shift
echo "arg1=$1, arg2=$2, arg3=$3"
shift
echo "arg1=$1, arg2=$2, arg3=$3"

% show_shift William Richard Elizabeth
arg1=William, arg2=Richard, arg3=Elizabeth
arg1=Richard, arg2=Elizabeth, arg3=
arg1=Elizabeth, arg2= , arg3=
```

Example 3

```
% my_copy dir_name filename1 filename2 filename3 …

# This shell script copies all the files to
  directory "dir_name"

% cat my_copy
#!/bin/sh
# Script allows user to specify, as the 1st argument,
# the directory where the files are to be copied.
location=$1
shift
files=$*
cp $files $location
```

Shifting Multiple Times

Shifting arguments three positions: 3 ways to write it

```
shift
shift
shift

shift; shift; shift

shift 3
```

Changing Values of Positional Parameters

- Positional parameters `$1, $2,` … normally store command line arguments.
- Their values can be changed using the `set` command
- `set newarg1 newarg2 …`

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, environment variables have UPPER CASE and shell variables have lower 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        # current dir is home dir
funcex    # this fails because funcex
          # is in www/2031/Lecture9
set path=($path www/2031/Lecture9)
funcex    # successful
```

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

```
set path=($path .)
```

Readings

- Sections 3.6 to 3.8, UNIX textbook
- Chapter 5, UNIX textbook
- Posted tutorial on standard UNIX variables
- Posted Bourne shell tutorial

- Most importantly, play with the scripts we discussed in class