

Looping

- for *variable* in *list_of_items*
- do
- *command1*
- *command2*
- ...
- *last_command*
- done

Looping

- for filename in *
- do
- echo \$filename
- done
- for filename in *.doc
- do
- echo "Copying \$filename to \$filename.bak"
- cp \$filename \$filename.bak
- done

Looping

- for i in 1 2 3 4 5 6 7 8 9 10
- do
- echo -n "...\$i"
- done
- echo # Clean up for next shell prompt

Looping

```
# Counts by looping for a fixed number of times
# Note do on same line requires semicolon.
for i in 1 2 3 4 5 6 7 8 9 10; do
    echo -n "...$i"
done
echo   # Output newline
```

```
Counts by looping for a fixed number of times
# Note do on same line requires semicolon.
for i in 1 2 3 4 5 6 7 8 9 10; do
    echo -n "...$i"
done
sleep 5
echo   # Output newline
```

Looping

```
# Counts backwards
for i in 10 9 8 7 6 5 4 3 2 1
do
    echo -n "...$i"
done
echo      # Output new line
echo "Blast off!"
$ sh counter2
...10...9...8...7...6...5...4...3...2...1
Blast off!
```

Looping

```
for x in hello there world
do
echo next word is $x
done
```

- **Output:**
- **next word is hello**
- **next word is there**
- **next word is world**

Looping

- Use outputs of another program as the world list using back-quotes
- `for user in `cut -d: -f1 < /etc/passwd``
- `do`
- `echo -n "$user:"`
- `finger $user`
- `done`

Looping

```
# C-language-like for loop.  
# Must be run with bash.  
max=10  
for ((i=1; i <= max ; i++))  
do  
    echo -n "$i..."  
done  
echo
```

- `#!/usr/bin/env bash`
- `# cookbook filename: trackmatch`
- `#`
- `for CDTRACK in *`
- `do`
- `if [[ "$CDTRACK" =~ "([[:alpha:][:blank:]]*)-([[:digit:]]*)-(.*)$" ]]`
- `then`
- `echo Track ${BASH_REMATCH[2]} is ${BASH_REMATCH[3]}`
- `mv "$CDTRACK" "Track${BASH_REMATCH[2]}"`
- `fi`
- `done`

`bash 3.2`

- Ludwig Van Beethoven - 01 - Allegro.ogg
- Ludwig Van Beethoven - 02 - Adagio un poco mosso.ogg
- Ludwig Van Beethoven - 03 - Rondo - Allegro.ogg

Redirection

- What is that?
- `ls /fred > /dev/null 2> /dev/null`

While

- Example: reads every line from stdin
- `while read l`
- `do`
- `echo line: $l`
- `done`

While

- Reads from a file instead
 - `while read line`
 - `do`
 - `echo line: $line`
 - `done <file`
- Or run in subshell
 - `(while read line; do; echo line:$line; done) <file`

Until

- Same as “while” but test is inverted
 - `until test-cmd`
 - `do`
 - `command`
 - ...
 - `done`
- Execute ‘command’ while ‘test-cmd’ is false

Command line argument

- Special form of 'for' that implicitly loops over all
- command line arguments to script
- for x
- do
- ... x loops over all arguments ...
- done

Break and Continue

- Usual loop “escape” mechanisms:
- **break** [*n*]
- **continue** [*n*]
- The optional integer *n* specifies how many levels
- should be “broken” or “continued”.

case ... esac

- Like a “switch” statement
- **case** *string* **in**
- *expr1*)
- *command* ;;
- *expr2*)
- *command* ;;
- ...
- **esac**

case ... esac

- Note that each case must end with ‘;;’
- Unlike C: no ‘fall-through’
- Take first matching case then skip to bottom
- Expressions are special - wildcards plus ‘|’, where
- ‘|’ - means “or”
- **"hello|goodbye"** matches
- “hello” or “goodbye”

Example

- Mycal program In class discussion

```
#!/cs/local/bin/sh
```

```
case $# in
0) set `date`; m=$2; y=$6;;
1) m=$1; set `date`; y=$6;;
2) m=$1; y=$2;;
esac
```

```
tigger 212 % date
Wed Jan 28 14:38:38 EST 2009
tigger 213 %
```

```
case $m in
jan*|Jan*) m=1;;
feb*|Feb*) m=2;;
mar*|Mar*) m=3;;
apr*|Apr*) m=4;;
may*|May*) m=5;;
jun*|Jun*) m=6;;
jul*|Jul*) m=7;;
aug*|Aug*) m=8;;
sep*|Sep*) m=9;;
oct*|Oct*) m=10;;
nov*|Nov*) m=11;;
dec*|Dec*) m=12;;
[1-9]|10|11|12) ;;
*) y=$m;m="";;
esac
/usr/bin/cal $m $y
```

Shift

- Built-in command:
- **shift** “moves” all arguments down 1
- **\$2** is moved into **\$1**, **\$3** is moved into **\$2**, etc.
- **\$1** is thrown away
- shift generates an error if there are no arguments

Shift

- `#!/bin/sh`
- `while [ $# -gt 0 ]; do`
- `echo "processing arg: $1"`
- `shift`
- `done`

Shell Functions

- We can also write functions in the shell
- *name*() {
- commands ...
- }
- Parentheses '()' are always empty
- Note: no return type, no arguments declared
- Call via:
- *name* [*arg*] [*arg*] ...

Shell Functions

- Shell functions can take any number of arguments
- and return an exit status
- Arguments use the command-line argument syntax
- (**\$1**, **\$2**, ... etc)
- • **shift** works here as well!
- Return the exit status of the

Shell Functions

```
allfiles() {  
  # true if all args are files  
  for x in $*; do  
    if [ ! -f $x ]; then  
      return 1  
    fi  
  done  
  return 0  
}
```

Shell Functions

- A function must be defined before we can use it
- We call it like any other command
- **allfiles file1 dir1 file2**
- No parentheses in function call
- • Effectively creates a new (temporary) shell
- command

Shell Functions

Beware - there are no local variables!

```
x=      # set x to blank
```

```
f1() {  
x=hello  
}
```

```
f1 ; echo $x # call f1()
```

Output : "hello"

Solution: sub-shell (**f1**) or bash "**local**"

Environmental variables

- All shell variables are local to the shell
- To change any *environment* variable need to "**export**" them.
- Env. variables visible in programs via `etenv(3c)`.
- Syntax:
- `variable=value` -- normal assignment
- **export** *variable*
- Shell command "**env**" lists all env. variables