bash 3.x
Advanced Shell Scripting

Michael Potter

UniForum Chicago

October 24, 2006

Why bash?

- Simple to get started.
- Actively developed and ported.
- Includes advanced features.

Goals for Tonight

Understanding the Demos

The demo script

```
#!/opt/local/bin/bash
set -o option
echo "My process list:"    >outputfile.txt

ps -ef 2>&1 |grep "^$USR"  >outputfile.txt

echo "script finished: return code is $?"
```


pottmi@gmail.com

noclobber demo

```
localhost:~/presos/bash pottmi$ ./buggy.sh
script finished: return code is 0
localhost:~/presos/bash pottmi$ vim buggy.sh
localhost:~/presos/bash pottmi$ ./buggy.sh
./buggy.sh: line 4: outputfile.txt: cannot overwrite existing file
./buggy.sh: line 5: outputfile.txt: cannot overwrite existing file
script finished: return code is 1
localhost:~/presos/bash pottmi$ vim buggy.sh
localhost:~/presos/bash pottmi$ ./buggy.sh
./buggy.sh: line 5: outputfile.txt: cannot overwrite existing file
localhost:~/presos/bash pottmi$ vim buggy.sh
localhost:~/presos/bash pottmi$ ./buggy.sh
script finished: return code is 0
localhost:~/presos/bash pottmi$ cat ./buggy.sh
#!/opt/local/bin/bash

set -o noclobber
set -o errexit
mv outputfile.txt outputfile.bak
echo "My process list:"    >outputfile.txt
ps -ef 2>&1 |grep "^$USR" >>outputfile.txt
echo "script finished: return code is $?"

localhost:~/presos/bash pottmi$
```


pottmi@gmail.com

What did we learn?

- `set -o noclobber`
  - used to avoid overlaying files
- `set -o errexit`
  - used to exit upon error, avoiding cascading errors

command1 | command2

pipefail demo

```
localhost:~/presos/bash pottmi$ ./buggy.sh
script finished: return code is 0
localhost:~/presos/bash pottmi$ vim ./buggy.sh
localhost:~/presos/bash pottmi$ ./buggy.sh
localhost:~/presos/bash pottmi$ vim ./buggy.sh
localhost:~/presos/bash pottmi$ ./buggy.sh
script finished: return code is 1
localhost:~/presos/bash pottmi$ vim ./buggy.sh
localhost:~/presos/bash pottmi$ ./buggy.sh
error at about 9
localhost:~/presos/bash pottmi$ vim ./buggy.sh
localhost:~/presos/bash pottmi$ ./buggy.sh
script finished: return code is 0
localhost:~/presos/bash pottmi$ cat ./buggy.sh
#!/opt/local/bin/bash

set -o noclobber
set -o errexit
set -o pipefail
trap 'echo error at about $LINENO' ERR
mv outputfile.txt outputfile.bak
echo "My process list:"    >outputfile.txt
ps aux 2>&1 |grep "^$USR"  >>outputfile.txt
echo "script finished: return code is $?"

localhost:~/presos/bash pottmi$
```

What did we learn?

- `set -o pipefail`
  - unveils hidden failures
- `set -o errexit`
  - can exit silently
- `trap` *`command`* `ERR`
  - corrects silent exits
- `$LINENO`
  - enhances error reporting

nounset demo

```
localhost:~/presos/bash pottmi$ ./buggy.sh
script finished: return code is 0
localhost:~/presos/bash pottmi$ vim buggy.sh
localhost:~/presos/bash pottmi$ ./buggy.sh
./buggy.sh: line 10: USR: unbound variable
error at about 10
localhost:~/presos/bash pottmi$ vim ./buggy.sh
localhost:~/presos/bash pottmi$ ./buggy.sh
script finished: return code is 0
localhost:~/presos/bash pottmi$ head -2 outputfile.txt
My process list:
pottmi     202   5.0 14.8   758588 155552  ??  S    Fri08AM 260:28.88 /Applicati
localhost:~/presos/bash pottmi$ cat ./buggy.sh
#!/opt/local/bin/bash

set -o noclobber
set -o errexit
set -o pipefail
set -o nounset
trap 'echo error at about $LINENO' ERR
mv outputfile.txt outputfile.bak
echo "My process list:"    >outputfile.txt
ps aux 2>&1 |grep "^$USER"  >>outputfile.txt
echo "script finished: return code is $?"

localhost:~/presos/bash pottmi$
```

What did we learn?

- `set -o nounset`
  - exposes unset variables

the final demo script

```
#!/opt/local/bin/bash

set -o noclobber
set -o errexit
set -o pipefail
set -o nounset
trap 'echo error at about $LINENO' ERR

mv outputfile.txt outputfile.bak
echo "My process list:"    >outputfile.txt
ps aux 2>&1 |grep "^$USER"  >>outputfile.txt

echo "script finished: return code is $?"
```

the final demo script

```
#!/opt/local/bin/bash


. ./stringent.sh || exit 1


mv outputfile.txt outputfile.bak
echo "My process list:"    >outputfile.txt
ps aux 2>&1 |grep "^$USER"  >>outputfile.txt

echo "script finished: return code is $?"
```

stringent.sh

```
# stringent.sh

set -o errexit
set -o noclobber
set -o nounset
set -o pipefail

function traperr
{
   echo "ERROR: ${BASH_SOURCE[1]} " \
        "at about line ${BASH_LINENO[0]}"
}

set -o errtrace
trap traperr ERR
```

BASH_SOURCE/BASH_LINENO

```
echo "ERROR: ${BASH_SOURCE[1]} " \
        "at about line ${BASH_LINENO[0]}"
```

```
ERROR: ./buggy.sh at about line 7
```

`${FUNCNAME[$i]}` was called at
`${BASH_LINENO[$i]}` in `${BASH_SOURCE[$i]}`

BASH_COMMAND

```
function traperr
{
    echo "ERROR: $1 ${BASH_SOURCE[1]} " \
         "at about line ${BASH_LINENO[0]}"
}

trap 'traperr $BASH_COMMAND' ERR
```

BROKEN

pottmi@gmail.com

PIPESTATUS

```
bash-3.1$  ps -ef 2>&1 |grep "^$USR" >/dev/null
bash-3.1$  echo "PIPESTATUS = ${PIPESTATUS[*]}  \$? = $?"
 PIPESTATUS = 1 0  $? = 0
bash-3.1$  set -o pipefail
bash-3.1$  ps -ef 2>&1 |grep "^$USR" >/dev/null
bash-3.1$  echo "PIPESTATUS = ${PIPESTATUS[*]}  \$? = $?"
 PIPESTATUS = 1 0  $? = 1
bash-3.1$  ps aux 2>&1 |grep "^$USER" >/dev/null
bash-3.1$  echo "PIPESTATUS = ${PIPESTATUS[*]}  \$? = $?"
 PIPESTATUS = 0 0  $? = 0
bash-3.1$  echo "PIPESTATUS = ${PIPESTATUS[*]}  \$? = $?"
 PIPESTATUS = 0  $? = 0
```

Variables

Integer Demo

```
bash-3.1$ cat ./integer.sh
#!/opt/local/bin/bash

MyInt=0
echo "Value = $(( ($MyInt+5) * 2 ))"

bash-3.1$ ./integer.sh
Value = 10
bash-3.1$ vim ./integer.sh
bash-3.1$ ./integer.sh
./integer.sh: line 5: 0: unbound variable
bash-3.1$ vim ./integer.sh
bash-3.1$ ./integer.sh
./integer.sh: line 4: 0: unbound variable
bash-3.1$ vim ./integer.sh
bash-3.1$ ./integer.sh
Value = 10
bash-3.1$ cat ./integer.sh
#!/opt/local/bin/bash
. ./stringent.sh

declare -i MyInt=0
echo "Value = $(( ($MyInt+5) * 2 ))"

bash-3.1$
```

What did we learn

- `stringent.sh`
  - Proven to be a good idea
- `declare -i` *`variable`*
  - non-integer values caught sooner
- unset variables used as an int are 0
  - unless caught with `set -o nounset`
- `$(( ... ))`
  - arithmetic syntax

gotcha

```
declare -i MyInt1=012
declare -i MyInt2=0x12

echo "Value1 = $MyInt1"
echo "Value2 = $MyInt2"
printf "%o %x\n" $MyInt1 $MyInt2

Value1 = 10
Value2 = 18
12 12
```

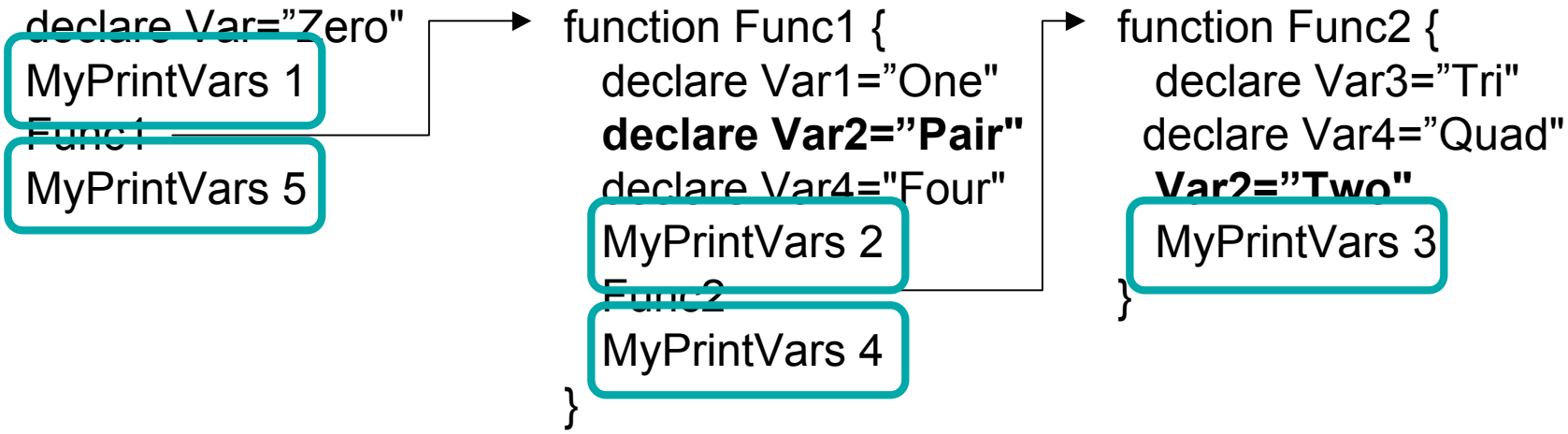
Arithmetic Syntax

- intA=$((($intB + 5) * 2))
  - Allowed anywhere a variable is allowed
- let "intA = ($intB + 5) * 2"
  - returns 0 or 1
- ((intA = ($intB + 5) * 2))
  - equivalent to let
- intA=\($intB+5\)*2
  - no spaces allowed
  - Special characters must be escaped
  - intA must be declare -i
- intA=$[($intB + 5) * 2]
  - deprecated

local variables

- weak
- good enough
- not just local, local and below
- two ways to declare:
  - declare
  - local
- $1, $2, ... are not scoped the same

Scoping

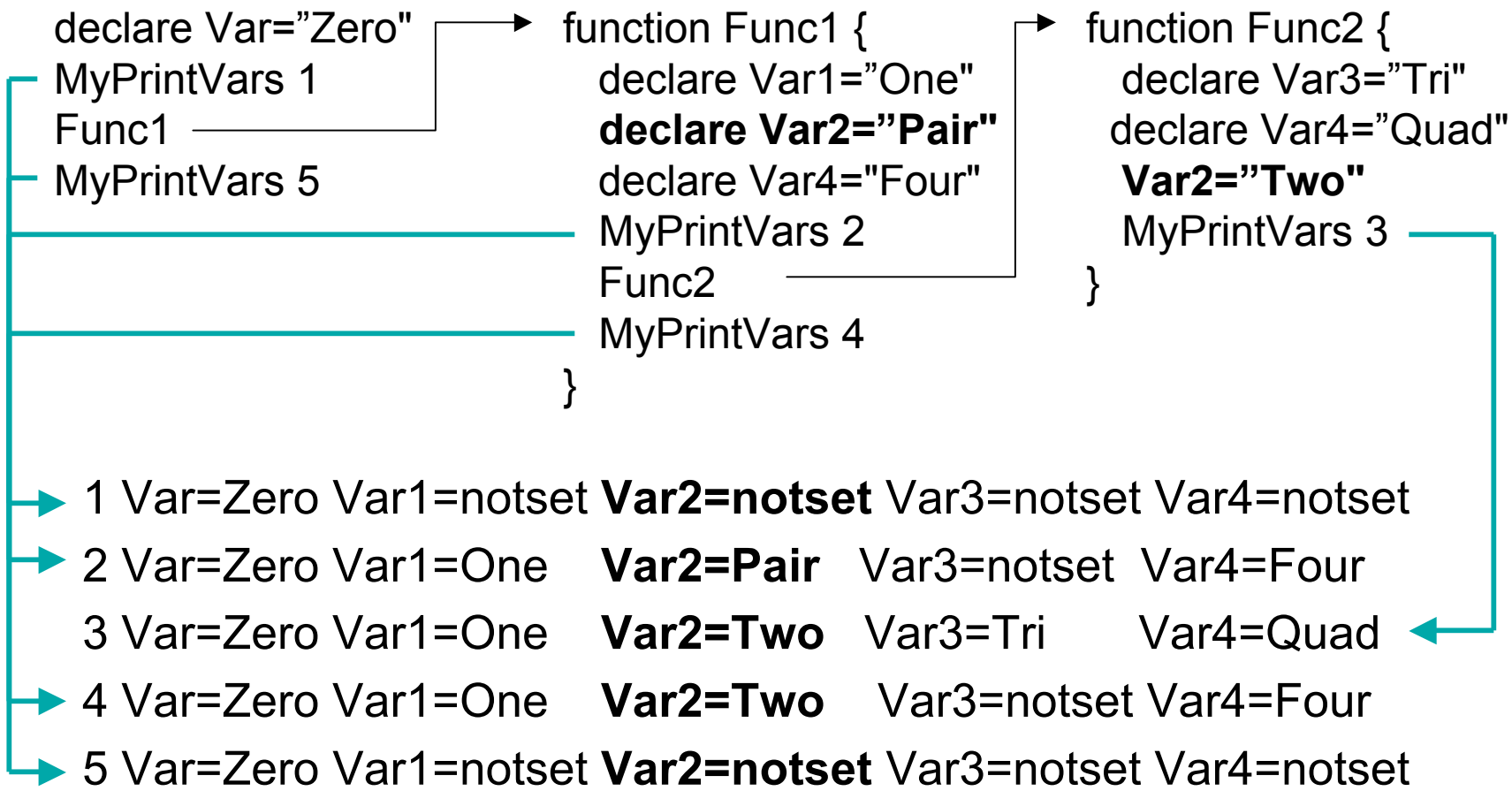


pottmi@gmail.com

Handling undefined variables

```
function PrintVars {
   echo -n "Var1=${Var1:-notset}"
   echo -n "Var2=${Var2:-notset}"
   echo -n "Var3=${Var3:-notset}"
   echo -n "Var4=${Var4:-notset}"
   echo    "Var5=${Var5:-notset}"
}
```

Scoping



pottmi@gmail.com

readonly variables

- Two ways to declare
  - declare -r
  - readonly
- One way trip
- Used with -i to create readonly integers
- readonly can be used on system variables
  - e.g. keep users from changing their prompt
  - not documented!

conditionals

if *command*

if (( ))

if [ ]

if test

if [[ ]]

if *command*

```
set +o errexit
grep Jim /etc/passwd
declare -i Status=$?
set -o errexit
if (( $Status == 0 ))
then
   echo “Jim is a user”
fi
```

```
if grep Jim /etc/passwd
then
   echo “Jim is a user”
fi
```

```
! grep Jim /etc/passwd
declare -i Status=$?
if (( $Status != 0 ))
then
   echo “Jim is a user”
fi
```

What did we learn?

- set +o errexit turns off errexit
- Save $? to a permanent variable
- ! turns off errexit for a single command
- zero is true, non-zero is false
- if (( )) used for numeric tests

gotcha

- if [[$Age > 21]] # bad
  - > is a string comparison operator
- if [$Age > 21] # bad
  - > is a redirection operator
- if [[$Age -gt 21]] # good
  - fails in strange ways if $Age is not numeric
- if (($Age > 21)) # best
  - $ on Age is optional

pottmi@gmail.com

test and [

```
bash-3.1$ which test
/bin/test
bash-3.1$ which [
/bin/[
bash-3.1$ ls -l /bin/[ /bin/test
-r-xr-xr-x 2 root wheel 18104 Aug 21 2005 /bin/[
-r-xr-xr-x 2 root wheel 18104 Aug 21 2005 /bin/test
```

So?

if [[ ]]

[versus [[

- `[[ $a == z* ]]`
  - True if $a starts with an "z”.
- `[[ $a == "z*" ]]`
  - True if $a is exactly equal to “z*”.
- `[ $a == z* ]`
  - Error if $a has a space.
  - Error if more than one filename starts with z.
  - True if a filename exists that starts with z and is exactly $a.
  - True if no filenames exist that start with z and $a equals z*.
- `[ "$a" == "z*" ]`
  - True if $a is exactly equal to z*.

pottmi@gmail.com

the rules

- use [
  - when you “want” to use file globbing
- use ((
  - when you want to do math
- use [[
  - for everything else

regular expressions

- Introduced with version 3.0
- Implemented as part of [[ ]]
- Uses binary operator =~
- Supports extended regular expressions
- Supports parenthesized subexpressions

regular expression

```
declare MyStr="the quick brown fox"

[[ $MyStr == "the*" ]]    # false: must be exact
[[ $MyStr == the* ]]      # true: pattern match

[[ $MyStr =~ "the" ]]     # true
[[ $MyStr =~ the ]]       # true
[[ $MyStr =~ "the*" ]]    # true
```

subexpressions

```
declare MyStr="the quick brown fox"

if [[ $MyStr =~ "the ([a-z]*) ([a-z]*)" ]]
then
   echo "${BASH_REMATCH[0]}” # the quick brown
   echo "${BASH_REMATCH[1]}” # quick
   echo "${BASH_REMATCH[2]}” # brown
fi
```

bad expressions

```
declare MyStr="the quick brown fox"

if [[ $MyStr =~ "the [a-z) ([a-z*)" ]]
then
   echo "got a match"
elif (( $? == 2 ))
then
   traperr "Assertion Error: Regular expression error"
   exit 1
fi
```

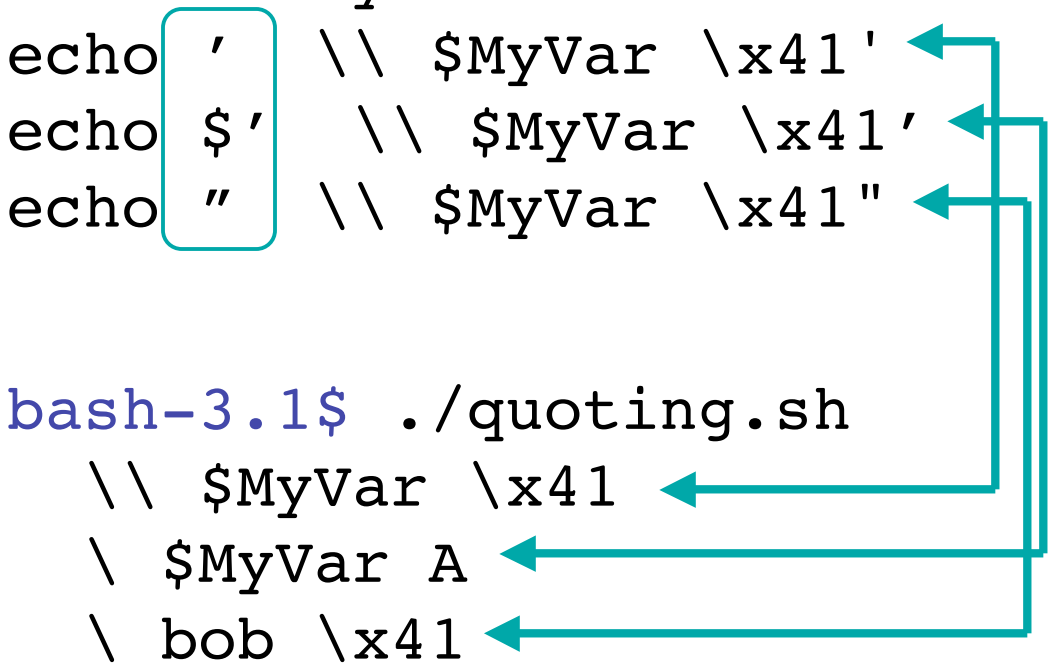
gotcha

- `cp $srcfile $dstfile`
  - broken if $srcfile has a space
- `cp “$srcfile” “$dstfile”`
  - broken if srcfile begins with -
- `cp -- “$srcfile” “$dstfile”`

quoting

```
declare MyVar="bob"
echo  '   \\ $MyVar \x41'
echo  $'   \\ $MyVar \x41'
echo  "   \\ $MyVar \x41"


bash-3.1$ ./quoting.sh
   \\ $MyVar \x41
   \ $MyVar A
   \ bob \x41
```



quoting recommendation

- quote variables liberally
  - extra quotes likely to cause a consistent error
  - missing quotes are likely to cause inconsistent behavior
- Safe Exceptions
  - within if [[ ]]
  - Integer variables (define -i)
  - within if (( ))

Handling undefined variables

```
function PrintVars {
   echo -n "Var1=${Var1:-notset}"
   echo -n "Var2=${Var2:-notset}"
   echo -n "Var3=${Var3:-notset}"
   echo -n "Var4=${Var4:-notset}"
   echo -n "Var5=${Var5:-notset}"
}
```


pottmi@gmail.com

unset variables

- `${parameter -word}`
  - returns word
- `${parameter +word}`
  - returns empty (returns word if set)
- `${parameter =word}`
  - sets parameter to word, returns word
- `${parameter ?message}`
  - echos message and exits

unset variables

- `${parameter-word}`
- `${parameter+word}`
- `${parameter=word}`
- `${parameter?message}`

default variables

```
function MyDate
{
   declare -i Year=${1:?"$0 Year is required"}
   declare -i Month=${2:-1}
   declare -i Day=${3:-1}

   if (( $Month > 12 )); then
      echo "Error Month > 12" >&2
      exit 1
   fi
   if (( $Day > 31 )); then
      echo "Error Day > 31" >&2
      exit 1
   fi

   echo "$Year-$Month-$Day"
}
```

sub strings

```
declare MyStr="The quick brown fox"

echo "${MyStr:0:3}"   # The
echo "${MyStr:4:5}"   # quick
echo "${MyStr: -9:5}" # brown
echo "${MyStr: -3:3}" # fox
echo "${MyStr: -9}"   # brown fox
```


pottmi@gmail.com

substr by pattern

- `${Var#pattern}`
- `${Var%pattern}`
- `${Var##pattern}`
- `${Var%%pattern}`

a jingle

We are #1 because we give 110%

substr by pattern

```
declare MyStr="/home/pottmi/my.sample.sh"

echo "${MyStr##*/}" # my.sample.sh
echo "${MyStr%.*}"  # /home/pottmi/my.sample
echo "${MyStr%/*}"  # /home/pottmi

echo "${MyStr#*/}" #home/pottmi/my.sample.sh
echo "${MyStr%%.*}" # /home/pottmi/my
```


pottmi@gmail.com

search and replace

- `${Var/pattern/replace}`

substr by pattern

```
declare MyStr="the fox jumped the dog"

echo "${MyStr/the/a}"
                                  # a fox jumped the dog
echo "${MyStr//the/a}”
                                  # a fox jumped a dog
echo "${MyStr//the }”
                                  # fox jumped dog
```

unintended subshells

```
declare -i Count=0
declare Lines

cat /etc/passwd | while read Lines
do
   echo -n "."
  ((Count++))
done

echo " final count=$Count"
```

..... final count=0

unintended subshells

```
declare -i Count=0
declare Lines

                    while read Lines
do
   echo -n "."
  ((Count++))
done </etc/passwd

echo " final count=$Count"
```

..... final count=38

unintended subshells

```
declare -i Count=0
declare Lines

                    while read Lines
do
   echo -n "."
  ((Count++))
done < <(cat /etc/passwd)

echo " final count=$Count"
```

..... final count=38

Learn more

- man bash
- O'Reilly - 'Learning the Bash shell'
- http://bashdb.sourceforge.net/bashref.html
- http://www.faqs.org/docs/abs/HTML/
- Ask me to help!

Contact Information

Michael Potter
PO Box 469
Lisle, IL 60532

+1 630 926 8133

pottmi@gmail.com

Copyright Notice