Scripting versus Programming

bash supports a scripting language.

Programming languages are generally a lot more powerful and a lot faster than scripting languages.

Programming languages generally start from source code and are compiled into an executable. This executable is not easily ported into different operating systems.

A scripting language also starts from source code, but is not compiled into an executable. Rather, an interpreter reads the instructions in the source file and executes each instruction.

Interpreted programs are generally slower than compiled programs.

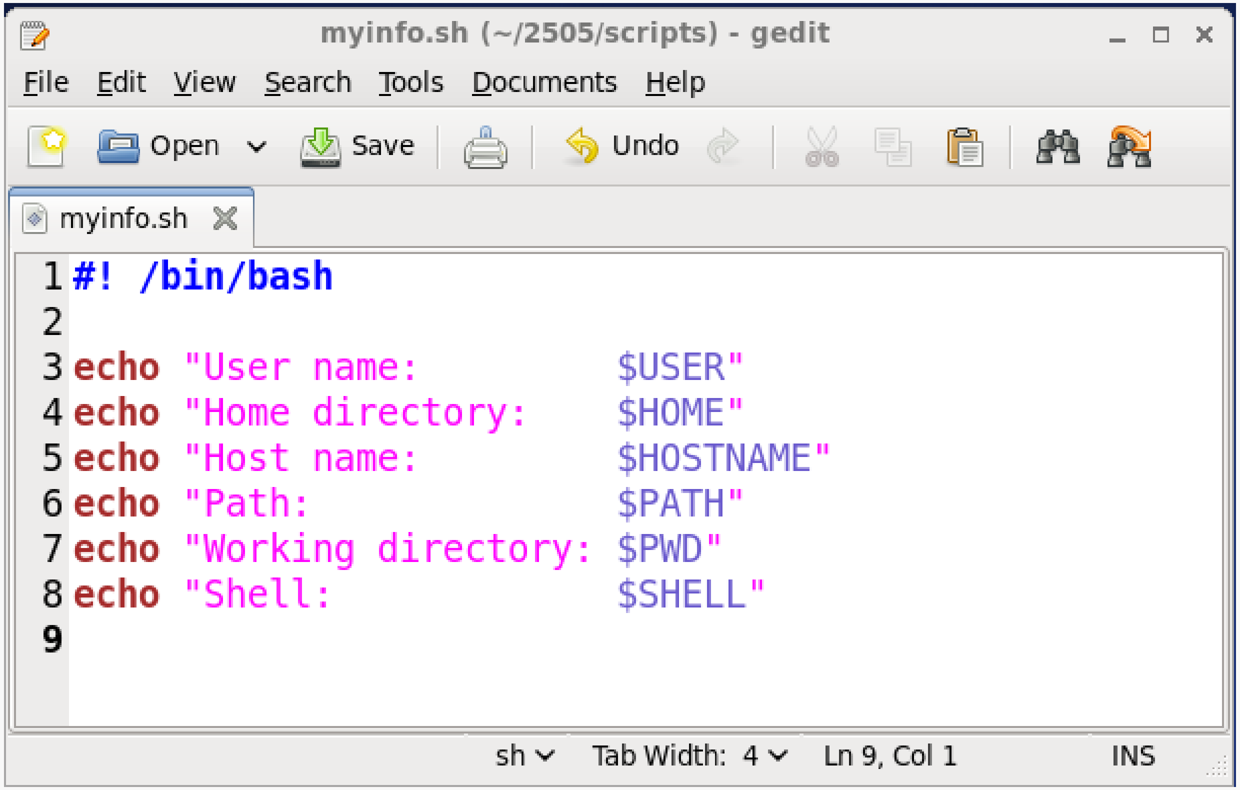
The main advantage is that you can easily port the source file to any operating system. bash is a scripting language. Some other examples of scripting languages are Perl, Lisp, and Tcl.

First Example

bash scripts are just text files (with a special header line) that contain commands.

We recommend using the extension "sh" when naming script files.

You can create a script using any text editor:



myinfo.sh (~/2505/scripts) - gedit
File Edit View Search Tools Documents Help
Open
Save
Undo
myinfo.sh
sh
Tab Width: 4
Ln 9, Col 1
INS

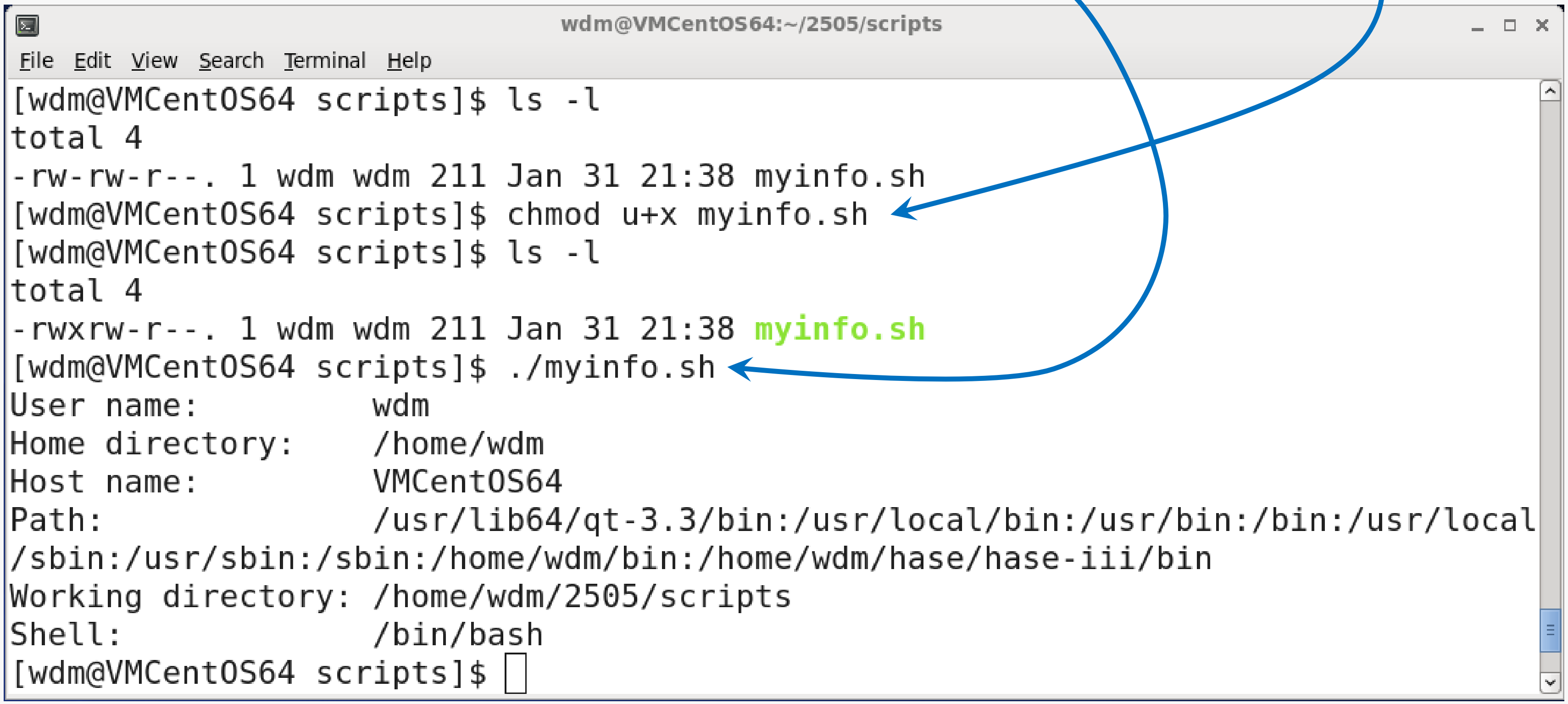
```
#! /bin/bash

echo "User name:         $USER"
echo "Home directory:    $HOME"
echo "Host name:         $HOSTNAME"
echo "Path:              $PATH"
echo "Working directory: $PWD"
echo "Shell:             $SHELL"

```

myinfo.sh

Running the Script

To execute the script you must first set execute permissions (see below).

Then, just invoke the script as a command, by name:

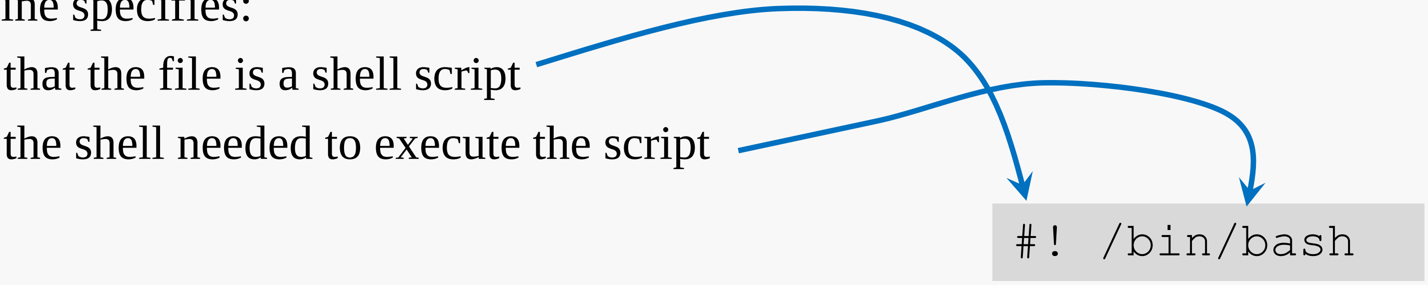


wdm@VMCentOS64:~/2505/scripts
File Edit View Search Terminal Help

```
[wdm@VMCentOS64 scripts]$ ls -l
total 4
-rw-rw-r--. 1 wdm wdm 211 Jan 31 21:38 myinfo.sh
[wdm@VMCentOS64 scripts]$ chmod u+x myinfo.sh
[wdm@VMCentOS64 scripts]$ ls -l
total 4
-rwxrw-r--. 1 wdm wdm 211 Jan 31 21:38 myinfo.sh
[wdm@VMCentOS64 scripts]$ ./myinfo.sh
User name:         wdm
Home directory:    /home/wdm
Host name:         VMCentOS64
Path:              /usr/lib64/qt-3.3/bin:/usr/local/bin:/usr/bin:/bin:/usr/local
/sbin:/usr/sbin:/sbin:/home/wdm/bin:/home/wdm/hase/hase-iii/bin
Working directory: /home/wdm/2505/scripts
Shell:             /bin/bash
[wdm@VMCentOS64 scripts]$ 
```

Analysis

The first line specifies:



that the file is a shell script

the shell needed to execute the script

```
#! /bin/bash
```

echo writes a line of text to standard output.

```
echo "User name:         $USER"
echo "Home directory:    $HOME"
echo "Host name:         $HOSTNAME"
echo "Path:              $PATH"
echo "Working directory: $PWD"
echo "Shell:             $SHELL"
```

`USER` is a global variable maintained by the bash shell; it stores the user name of whoever's running the shell.

`$` causes the variable `USER` to be *expanded* (replaced with its value).

Variables

You may create variables local to your shell by simply using them:

`VARNAME="value"`

```
#! /bin/bash

message="Hello, world!"
echo $message
```

Variable names are case-sensitive, alphanumeric, and may not begin with a digit.

bash reserves a number of global variable names for its own use, including:

```
PATH        HOME        CDPATH
PS1         PS2         LANG
```

See the references for a complete list and descriptions.

Variables

By default, script variables can store any value assigned to them.

Typically variables are used to hold strings or integers.

```
#! /bin/bash

one=1
two=2
three=$((one + two))  # syntax forces arith. expansion

echo $one
echo $two
echo $three
```

Spaces are not allowed around the assignment operator.

A Script for Backups

backup.sh adapted from [2]

```
#!/bin/bash
# This script makes a backup of my ~/2505 directory.
# Change the variables to make the script work for you:

BACKUPDIR=$HOME/2505            # directory to be backed up
TARFILE=/var/tmp/2505.tar       # tar file created during backup
SERVER=ap1.cs.vt.edu            # server to copy backup to
REMOTEID=wmcquain               # your ID on that server
REMOTEDIR=/home/staff/wmcquain  # dir to hold  backup on server
LOGFILE=~/logs/2505_backup.log  # local log file recording backups

# Move into the directory to be backed up
cd $BACKUPDIR

# Run tar to create the archive.
tar cf $TARFILE *

# Copy the file to another host.
scp $TARFILE $REMOTEID@$SERVER:$REMOTEDIR

# Create a timestamp in the logfile to record the backup operation.
date >> $LOGFILE
echo backup succeeded >> $LOGFILE

exit 0                           # return 0 on success
```

A Script for Backups

```
bash > ./backup.sh
wmcquain@ap1.cs.vt.edu's password:
2505.tar                      100%   30KB  30.0KB/s   00:00
```

The script is missing some desirable features:

- the ability to specify the directory to be backed up on the command-line
- error-checking to be sure that directory exists
- checking the exit codes for the various commands called by the script

We may add some of those features later...

Special Variables

There are some special variables that can be referenced but not assigned to.

The following is incomplete and somewhat oversimplified:

`$*`	used to access the positional command-line parameters
`$@`	used to access the positional command-line parameters
`$#`	expands to the number of positional parameters
`$?`	expands to the exit status of the most recently executed command
`$k`	(k an integer) the k-th positional command-line parameter

```
#! /bin/bash

echo "There were $# parameters!"
echo "$@"
```

Special Variables

The ability to catch the exit code from a command is useful in detecting errors:

```
#! /bin/bash

ls -e *
exitcode="$?"

echo "ls exited with: $exitcode"
```

The UNIX convention is that 0 is returned on success and nonzero on failure.

From the man page for ls:

Exit status:

0 if OK,

1 if minor problems (e.g., cannot access subdirectory),

2 if serious trouble (e.g., cannot access command-line argument).

Escape Characters

The backslash character (outside of quotes) preserves the literal value of the next character that follows it:

```
bash > today=20140201
bash > echo $today
20140201
bash > echo \$today
$today
```

BTW, note that this also shows we can apply variables from the command prompt.

Single Quotes and Double Quotes

Single quotes preserve the literal value of every character within them:

```
bash > echo '$today'
$today
```

Double quotes preserve the literal value of every character within them except the dollar sign $, backticks ` `, and the backslash \:

```
bash > echo "$today"
20140201
```

Shell Expansion: Braces

An expression of the form

`preamble{comma-separated-list}postfix`

expands to a sequence of values obtained by concatenating the preamble and postscript with each element in the comma-separated list within the braces:

```
bash > echo eleg{ant,aic,ible}
elegant elegaic elegible
```

Command Expansion

We can replace a command with its output by using either:

`` `command` `` or **`$(command)`**

```
bash > echo date
date
bash > echo `date`
Sat Feb 1 19:52:08 EST 2014
bash > echo $(date)
Sat Feb 1 19:53:17 EST 2014
```

Arithmetic Expansion

Arithmetic computations can be carried out directly, using the syntax for arithmetic expansion:

```
$((expression))
```

Arithmetic computations can be carried out directly, using the syntax for arithmetic expansion.

The available operators are shown on the next slide.

The usual C-like precedence rules apply, but when in doubt, parenthesize.

Leading 0 denotes an octal value; leading 0X a hexadecimal value.

Arithmetic Operators

Operator	Meaning
VAR++ and VAR--	post-increment and post-decrement
++VAR and --VAR	pre-increment and pre-decrement
- and +	unary minus and plus
! and ~	logical and bitwise negation
**	exponentiation
*, / and %	multiplication, division, remainder
+ and -	addition, subtraction
<< and >>	left and right bitwise shifts
<=, >=, < and >	comparison operators
== and !=	equality and inequality
&	bitwise AND
^	bitwise exclusive OR
|	bitwise OR
&&	logical AND
||	logical OR
expr ? expr : expr	conditional evaluation
=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^= and |=	assignments
,	separator between expressions

Example

add.sh

```
#! /bin/bash

left=$1      # left gets parameter 1
right=$2     # right gets parameter 2

sum=$((left + right))  # sum gets result of addition

echo "$0 says the sum of $left and $right is $sum."
exit 0
```

```
bash > ./add.sh 83231 70124
./add.sh says the sum of 83231 and 70124 is 153355.
```

The example lacks a conditional check for the number of parameters; we will fix that a bit later...

Control Structures: if/then

bash supports several different mechanisms for selection; the most basic is:

```
. . .
if [[ condition ]]; then

   commands     # executed iff condition eval to true
fi
. . .
```

Be careful about the syntax here.

The spaces after "[[" and before "]]" are required, as is the semicolon!

NB: there is an older notation using single square brackets; for a discussion see:

http://mywiki.wooledge.org/BashFAQ/031

Example

We can fix one problem with the adder script we saw earlier by adding a check on the number of command-line parameters:

add2.sh

```
#! /bin/bash

if [[ $# -ne 2 ]]; then
   echo "Invocation:  ./add.sh integer integer"
   exit 1
fi

left=$1
right=$2

sum=$((left + right))

echo "$0 says the sum of $left and $right is $sum."
```

NB: integers are compared using -gt, -lt, -ge, -le, -eq, -ne

But we could make it better if we could process a variable number of command-line parameters... that will come a bit later...

Control Structures: if variations

```
. . .
if [[ condition ]]; then

   commands executed if condition evaluates true
else
   commands executed if condition evaluates false
fi
. . .
```

```
. . .
if [[ condition1 ]]; then

   commands   // condition1
elif [[ condition2 ]]; then
   commands   // !condition1 && condition2
. . .
else
   commands   // !condition1 && !condition2 &&...
fi
. . .
```

Example

add3.sh

```
#! /bin/bash

if [[ $# -lt 2 || $# -gt 4 ]]; then
   echo "Invocation:  ./add.sh integer integer [integer [integer]] "
   exit 1
fi

if [[ $# -eq 2 ]]; then
   echo "$0 says the sum of $1 and $2 is $(($1 + $2))."
elif [[ $# -eq 3 ]]; then
   echo "$0 says the sum of $1, $2 and $3 is $(($1 + $2 + $3))."
else
   echo "$0 says the sum of $1, $2, $3 and $4 is $(($1 + $2 + $3 + $4))."
fi

exit 0
```

Aside: File-related Tests

There are a number of expressions that can be used within the braces for the conditional, for testing files, including:

`-e FILE`	true iff `FILE` exists
`-d FILE`	true iff `FILE` exists and is a directory
`-r FILE`	true iff `FILE` exists and is readable
`-w FILE`	true iff `FILE` exists and is writeable
`-x FILE`	true iff `FILE` exists and is executable
`-s FILE`	true iff `FILE` exists and has size > 0

The logical operator `!` (not) can be prefixed to these tests.

Aside: String-related Tests

There are a number of expressions that can be used within the braces for the conditional, for testing strings, including:

`-z STRING`	true iff `STRING` has length zero
`-n STRING`	true iff `STRING` has length greater than zero

Strings may be compared via the following tests:

`STR1 == STR2`	true iff `STR1` equals `STR2`
`STR1 != STR2`	true iff `STR1` does not equal `STR2`
`STR1 < STR2`	true iff `STR1` precedes `STR2`
`STR1 > STR2`	true iff `STR1` succceeds `STR2`

Aside: Integer-related Tests

There are a number of expressions that can be used within the braces for the conditional, for testing integers, including:

`I1 -eq I2`	true iff `I1` == `I2`
`I1 -ne I2`	true iff `I1 !=` `I2`
`I1 -lt I2`	true iff `I1` < `I2`
`I1 -le I2`	true iff `I1` <= `I2`
`I1 -gt I2`	true iff `I1` > `I2`
`I1 -ge I2`	true iff `I1` >= `I2`

Revised Script for Backups

backup2.sh adapted from [2]

```
#!/bin/bash
# This script makes a backup of a directory to another server.
# Invocation:  ./backup2.sh DIRNAME

if [[ $# -ne 1 ]]; then
   echo "Invocation:  ./backup2.sh DIRNAME"
   exit 1
fi

if [[ ! -d $1 ]]; then
   echo "$1 is not a directory"
   exit 2
fi

BACKUPDIR=$1                       # directory to be backed up

# Change the values of the variables to make the script work for you:
TARFILE=/var/tmp/mybackup.tar      # tar file created during backup
SERVER=ap1.cs.vt.edu               # server to copy backup to
REMOTEID=wmcquain                  # your ID on that server
REMOTEDIR=/home/staff/wmcquain      # dir to hold  backup on server
LOGFILE=~/logs/backup.log          # local log file recording backups
. . .
```

verify there is a command-line parameter
and
that it names a directory

Revised Script for Backups

backup.sh adapted from [2]

```
. . .
# Move into the directory to be backed up
cd $BACKUPDIR

# Run tar to create the archive.
tar cf $TARFILE *
if [[ $? -ne 0 ]]; then
   echo "Aborting: tar returned error code $?"
   exit 3
fi

# Copy the file to another host.
scp $TARFILE $REMOTEID@$SERVER:$REMOTEDIR
if [[ $? -ne 0 ]]; then
   echo "Error: scp returned error code $?"
   exit 4
fi

# Create a timestamp in the logfile to record the backup operation.
echo "$BACKUPDIR:  `date`" >> $LOGFILE

exit 0                             # return 0 on success
```

check exit code from tar

check exit code from scp

Control Structures: while/do

bash supports several different mechanisms for iteration, including:

```
. . .
while [[ condition ]]; do

   commands     # executed iff condition eval to true
done
. . .
```

Example

gcd.sh

```
#! /bin/bash
if [[ $# -ne 2 ]]; then
   echo "Invocation:  ./gcd.sh integer integer"
   exit 1
fi

# Apply Euclid's Algorithm to find GCD:
x=$1
y=$2
# Operands need to be non-negative:
if [[ x -lt 0 ]]; then x=$((-x))
fi
if [[ y -lt 0 ]]; then y=$((-y))
fi

while [[ y -gt 0 ]]; do
   rem=$(($x % $y))
   x=$y
   y=$rem
done

# Report GCD:
echo "GCD($1, $2) = $x"

exit 0
```

Control Structures: for/do

```
. . .
for VALUE in LIST; do

   commands      # executed on VALUE
done
. . .
```

```
for x in one two three four; do
   str+=" $x"
   echo "$str"
done
```

```
list="one two three four"

for x in $list; do
   str+=" $x"
   echo "$str"
done
```

for1.sh

Parameter List and for/do

add4.sh

```
#! /bin/bash

sum=0
if [[ $# -eq 0 ]]; then
   echo "Nothing to add"
   exit 1
fi

for x; do
   echo "     $x"
   sum=$(($sum + $x));
done
echo "sum: $sum"

exit 0
```

NB: if you omit "in LIST", it defaults to "in $@", which is the positional parameter list

```
bash > ./add4.sh 17 13 5 8 10 73
     17
     13
     5
     8
     10
     73
sum: 126
```

Functions

bash supports defining functions that scripts can call.

A function simply groups a collection of instructions and gives the collection a name.

Parameters may be passed, but in the manner they're passed to a script by the command shell – the syntax is not what you are used to.

The implementation of a function must occur before any calls to the function.

Variables defined within a function are (by default) accessible outside (after) the function definition – that’s not what you are used to.

Two syntaxes:

```
function funcname {
    commands
}
```

```
funcname() {
    commands
}
```

Defining a Function

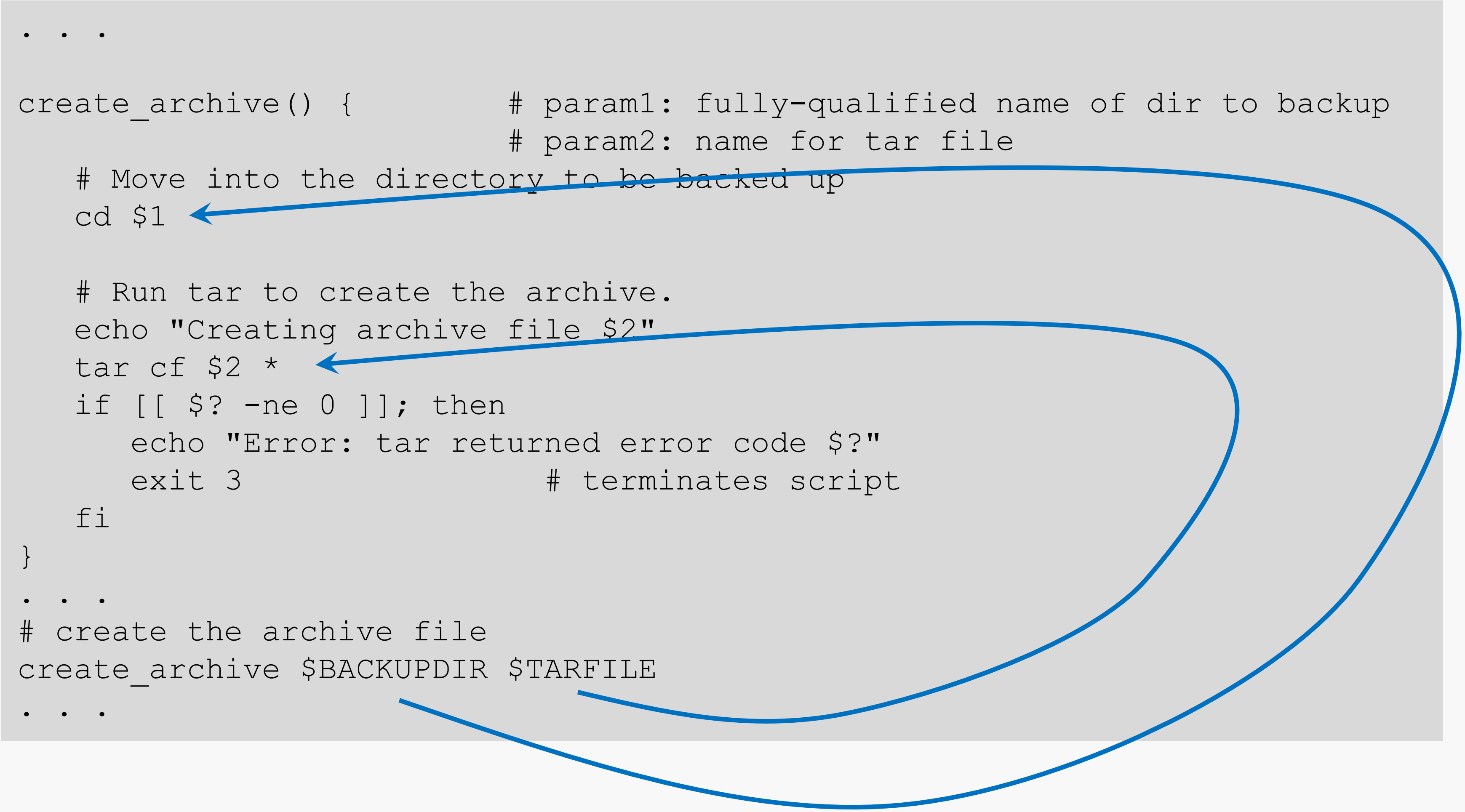
In the backup script, we have the following block of code to create the archive file:

```
. . .
# Move into the directory to be backed up
cd $BACKUPDIR

# Run tar to create the archive.
tar cf $TARFILE *
if [[ $? -ne 0 ]]; then
   echo "Aborting: tar returned error code $?"
   exit 3
fi
. . .
```

We can wrap this into a function interface, and take the name of the directory to be backed up and the name to give the tar file parameters to the function…

Defining and Calling a Function

We can wrap this into a function interface, and take the name of the directory to be backed up and the name to give the tar file parameters to the function…



```
. . .

create_archive() {        # param1: fully-qualified name of dir to backup
                          # param2: name for tar file
   # Move into the directory to be backed up
   cd $1

   # Run tar to create the archive.
   echo "Creating archive file $2"
   tar cf $2 *
   if [[ $? -ne 0 ]]; then
      echo "Error: tar returned error code $?"
      exit 3                # terminates script
   fi
}
. . .
# create the archive file
create_archive $BACKUPDIR $TARFILE
. . .
```

Backup Script with Functions

```
#!/bin/bash
# This script makes a backup of a directory to another server.
# Invocation:  ./backup3.sh DIRNAME

#################################################### fn definitions
show_usage() {
   echo "Invocation:  ./backup2.sh DIRNAME"
}

get_directory_name() {         # param1: fully-qualified name of dir to
backup
   P1=$1
   DIRNAME=${P1##*/}          # HERE BE DRAGONS!
}

set_variables() {
   # Change the values of the variables to make the script work for you:
   TARFILE=/var/tmp/$DIRNAME.tar      # tar file created during backup
   SERVER=ap1.cs.vt.edu               # server to copy backup to
   REMOTEID=wmcquain                  # your ID on that server
   REMOTEDIR=/home/staff/wmcquain     # dir to hold  backup on server
   LOGFILE=~/logs/backup.log          # local log file recording backups
}
. . .
```

Backup Script with Functions

```
. . .
create_archive() {             # param1: fully-qualified name of dir to
backup
                               # param2: name for tar file
   # Move into the directory to be backed up
   cd $1

   # Run tar to create the archive.
   echo "Creating archive file $2"
   tar cf $2 *
   if [[ $? -ne 0 ]]; then
      echo "Error: tar returned error code $?"
      exit 3                # terminates script
   fi
}
. . .
```

Backup Script with Functions

```
. . .
copy_to_server() {          # param1: fully-qualified name of tar file
                            # param2: user name on server
                            # param3: network name of server
                            # param4: destination dir on server
   # Copy the file to another host.
   echo "Copying $1 to $3:$4"
   scp $1 $2@$3:$4
   if [[ $? -ne 0 ]]; then
      echo "Error: scp returned error code $?"
      exit 4                # terminates script
   fi
}
. . .
```

Backup Script with Functions

```
. . .
rm_archive() {              # param1: full-qualified name of tar file

   echo "Removing archive file $1"
   rm -f $1
   if [[ $? -ne 0 ]]; then
      echo "Error: rm returned error code $?"
      exit 4                # terminates script
   fi
}

log_backup() {
   echo "$1:  `date`" >> $2
}
. . .
```

Backup Script with Functions

```
. . .
#################################################### body of script

if [[ $# -ne 1 ]]; then   # check for a parameter
   show_usage
   exit 1
fi

if [[ ! -d $1 ]]; then    # see if it's a directory
   echo "$1 is not a directory"
   exit 2
fi

BACKUPDIR=$1              # directory to be backed up

# Get actual directory name (strip leading path info, if any)
get_directory_name $BACKUPDIR

# set environment for backup
set_variables
. . .
```

Backup Script with Functions

```
. . .
# create the archive file
create_archive $BACKUPDIR $TARFILE

# copy the archive file to the server
copy_to_server $TARFILE $REMOTEID $SERVER $REMOTEDIR

# clean up archive file
rm_archive $TARFILE

# Create a timestamp in the logfile to record the backup operation.
log_backup $BACKUPDIR $LOGFILE

exit 0                             # return 0 on success
```

Backup Script Execution

```
bash > ./backup3.sh ~/2505
Creating archive file /var/tmp/2505.tar
Copying /var/tmp/2505.tar to
ap1.cs.vt.edu:/home/staff/wmcquain
wmcquain@ap1.cs.vt.edu's password:
2505.tar          100%      90KB      90.0KB/s     00:00
Removing archive file /var/tmp/2505.tar
bash >
```

IMO, a good script provides the user with feedback about progress and success or failure.

Here Be Dragons

In the backup script we need to strip any path information from the front of the fully-qualified name for the directory to be backed up.

For example, we need to carry out the following transformation:

`/home/wdm/2505` → `2505`

Here's how we do it:

```
. . .
DIRNAME=${P1##*/}
. . .
```

Here's how it works:

- `"*/"` stands for an arbitrary number of characters followed by a forward slash.
- `"*/"` is expanded to match the longest part of `P1` that matches that pattern.
- In this case, it works out to be `"/home/wdm/"`.
- This longest match is removed from `P1`, leaving `"2505"` in this case.

Since the path prefix must end with a forward slash, this gives us exactly what we want.

See page 128 in [2] if you want more discussion.

Special Characters

There are many characters that have special meaning to the bash shell, including:

#	begins comment (to end of line)
$	causes expansion of the following character
\	causes following character to NOT be special
/	path separator AND division operator
`	command substitution
*	wildcard for file name expansion

A full discussion is available in Chapter 3 of [3].

These special characters may also occur in contexts, like input strings, in which we need them to retain their normal meanings...

Quoting: Double vs Single

Enclosing an expression in double quotes causes most, but not all, special characters to be treated literally:

```
bash > echo #702

bash > echo "#702"
#702
bash > echo 7$12
72
bash > echo "7$12"
72
```

Enclosing an expression in single quotes causes all special characters to be treated literally:

```
bash > echo '7$12'
7$12
```

It's usually good practice to enclose a variable evaluation in double quotes, since the variable may be a string that may contain special characters that are not supposed to be interpreted by the shell.

Transforming Strings

`${VAR:OFFSET:LENGTH}`

Take `LENGTH` characters from `$VAR`, starting at `OFFSET`.

```
bash > str=mairzydoatsanddozydoats
bash > echo $str
mairzydoatsanddozydoats
bash > echo ${str:6:5}
doats
bash > echo $str
mairzydoatsanddozydoats
```

`${VAR#WORD}`
`${VAR##WORD}`

If `WORD` matches a prefix of `$VAR`, remove the shortest (longest) matching part of `$VAR` and return what's left. `'%'` specifies a match at the tail of `$VAR`.

```
bash > echo ${str#mairzy}
doatsanddozydoats
bash > echo ${str%doats}
mairzydoatsanddozy
```

Transforming Strings

```
bash > var=/home/user/johokie/2505

bash > echo ${var%/*}
/home/user/johokie


bash > echo ${var%%/*}



bash > echo ${var#*/}
home/user/johokie/2505


bash > echo ${var##*/}
2505
```

%/* matched "/2505" at end

%%/* matched everything from the end

#*/ matched nothing at the front

##*/ matched "/home/user/johokie/"

Transforming Strings

```
bash > echo $var
/home/user/johokie/2505

bash > var2=$var/
bash > echo $var2
/home/user/johokie/2505/

bash > echo ${var2%/}
/home/user/johokie/2505
```

%/ matched "/" at end

NB: sometimes you get a path string from the command-line, and the user may or may not have put a `'/'` on the end...

Transforming Strings

`${VAR/TOREPLACE/REPLACEMENT}`

`${VAR//TOREPLACE/REPLACEMENT}`

Replace the first (all) occurrence(s) of `TOREPLACE` in `$VAR` with `REPLACEMENT`.

```
bash > echo $str
mairzydoatsanddozydoats

bash > echo ${str/doats/doates}
mairzydoatesanddozydoats

bash > echo ${str//doats/doates}
mairzydoatesanddozydoates

bash > echo $str
mairzydoatsanddozydoats
```

replaced 1st occurrence of "doats"

replaced both occurrences of "doats"

original is unchanged

Example: unpacktars

One problem I needed to solve was that I had a directory of tar files submitted by students, where each tar file contained the implementation of a program, perhaps consisting of many files:

```
bash > ls
aakallam.C3.11.tar  dnguy06.C3.6.tar    laura10.C3.1.tar   samm.C3.5.tar
adahan.C3.5.tar     domnap.C3.5.tar     lucase93.C3.12     sammugg.C3.4.tar
aemoore.C3.5.tar    dustinst.C3.7.tar   magiks.C3.8.tar    samp93.C3.13.tar
afritsch.C3.11.tar  elena.C3.5.tar      marcato.C3.5.tar   sarahn93.C3.1.tar
```

What I needed was to extract the contents of each student's submission to a separate directory, named using the PID field from the name of the student's submission.

I also had to be concerned about the possibilities (at least):

- A submission might not be a tar file.
- There might be an error when extracting a tar file.
- Neither I nor my TAs wanted to do this manually.

Of course, the solution was to write a shell script...

Design: unpacktars

The desired functionality led to some design decisions:

- Do not hard-wire any directory names.
- Optionally, let the target directory (holding the subdirectories for student submissions) in a different, user-specified directory than the one that holds the tar files.
- Do not require the target directory to exist already; if it does, do not clear it.
- Name the subdirectories using the student PIDs since those are unique and already part of the tar file names.
- Provide the user with sensible feedback if anything goes wrong.

unpacktars.sh: Verifying a File Type

```
#! /bin/bash
#
# Invocation:  unpacktars.sh tarFileDir extractionRoot
#
#    tarFileDir must name a directory containing tar files
#    tar file names are in the form fname.*.tar
#    extractionRoot is where the subdirs will go
#
# For each file in the specified tar file directory:
#    If the file is a tar file
#      - a directory named dirname/fname is created
#      - the contents of the tar file are extracted into dirname/fname
#

############################################# fn to check for tar file
#                 param1:  name of file to be checked
isTar() {

   mimeType=`file -b --mime-type $1`
   [[ $mimeType == "application/x-tar" ]]
}
. . .
```

-b: omit filename from output
--mime-type: compact output

unpacktars.sh: Extracting the PID

```
. . .
##################################### fn to extract PID from file name
#                  param1: (possibly fully-qualified) name of file
getPID() {

   fname=$1

   # strip off any leading path info
   fname=${fname##*/}




   # extract first token of file name
   spid=${fname%%.*}
}
. . .
```

`"##*/"`

remove longest leading string ending with `'/'`

`"%%.*"`

remove longest trailing string starting with `'.'`

unpacktars.sh: Processing the tar File

```
. . .
##################################### fn to extract tar file to subdir
#                 param1: root dir for subdirs
#                 param2: full name of file
processTar() {

   # set PID from file name
   getPID $2

   # create subdirectory for extracted files
   mkdir "$1/$spid"

   # extract tar contents to that directory
   tar -xf "$2" -C "$1/$spid"


   if [[ $? -ne 0 ]]; then
      echo "  Error extracting files from $2"
   fi
}
. . .
```

"-C"
specify destination dir

check exit code from tar

unpacktars.sh: Validating the Command Line

```
. . .
############################################################## body
if [[ $# -ne 2 ]]; then
   echo "Usage:  unpacktars.sh tarFileDir extractRoot"
   exit 1
fi

############################################# parameter check
# get 1st parameter; trim trailing '/'
srcdir=$1
srcdir=${srcdir%/}


# verify it's a directory name
if [[ ! -d "$srcdir" ]]; then
   echo "First argument must be a directory"
   exit 1
fi
. . .
```

`"%/"`

remove trailing `'/'`, if any

Directory holding tar files to be processed MUST already exist.

unpacktars.sh: Validating the Command Line

```
. . .
# get 2nd parameter; trim trailing '/'
trgdir=$2
trgdir=${trgdir%/}

if [[ ! -e "$trgdir" ]]; then
   echo "Creating $trgdir"
   mkdir "$trgdir"

elif [[ ! -d "$trgdir" ]]; then
   echo "Error:  $trgdir exists but is not a directory"
   exit 2

fi
. . .
```

Target directory may or may not already exist...

If it does not, create it.
This also detects a regular file with the specified name.

If a regular file exists with that name, we can't (safely) create a the directory.

unpacktars.sh: Processing the Directory

```
.  .  .
############################################# begin processing
echo "Processing files in $srcdir to $trgdir"

# iterate through files in the directory
for tfile in $srcdir/*
do
   # verify we have a regular file
   if [[ -f "$tfile" ]]; then

      # see if we have a tar file
      isTar $tfile
      if [[ $? -eq 0 ]]; then
         # process the tar file
         processTar $trgdir $tfile

      else
         # notify user of stray file
         echo "  Found non-tar file $tfile"
      fi
   fi
done

exit 0
```

`"tfile in $srcdir/*"`

This will iterate over the files that exist in the source directory.

`Bibliography

[1] A Practical Guide to Linux Commands, Editors, and Shell Programming, 2nd Ed, Mark G. Sobell, Pearson, 2010

[2] Bash Guide for Beginners, Machtelt Garrels, Version 1.11 (http://tldp.org/LDP/Bash-Beginners-Guide/html/index.html)

[3] Advanced Bash Scripting Guide, Mendel Cooper, Version 6.6 (http://tldp.org/LDP/abs/html/index.html)