UNIX/LINUX SHELL SCRIPT PROGRAMMING CONVENTIONS AND STYLE

Ian! D. Allen - idallen@idallen.ca - www.idallen.com
Winter 2013 - January to April 2013

COURSE HOME PAGE | COURSE OUTLINE | ALL WEEKS | PLAIN TEXT

THIS TERM IS FINISHED - MATERIAL MAY NOT BE UPDATED
- SEE CURRENT TERM FOR UPDATES

UPDATED: 2013-01-25 20:44 EST

TABLE OF CONTENTS

This Term Is Finished - Material may not be updated
- See current term for updates

1 Shell Script Programming

When writing programs (scripts are programs), you are not simply trying to "get it to work"; but, you are also (and most importantly in an academic setting) practicing and demonstrating good programming techniques. These techniques will make it possible for other people to read and understand your programs after you have written them.

Employers tell us that programming techniques are more important than the list of which languages you know. Do you have good style?

Good programming techniques include such things as:

- Correct documentation (using guidelines given here)
  - internal documentation (labelling, function headers, etc.)
  - external documentation (man pages, user guides, testing, etc.)
- Writing clean, simple, robust code: "Less code is better code"!
  - checking the return status of important commands used in the script
  - choosing the correct loop and branch control structures
  - rewriting the algorithm to minimize duplicated code
  - making it right before you make it clever or fast
- Using correct indentation, style, and mnemonic names
  - variable and file names reflect their use in the algorithm
- Using correct modularization and scope (i.e. local vs. global variables)
- Correct use of arguments and/or prompting for user input
  - some scripts will prompt the user for input if no arguments are given
  - some scripts will read standard input if no file names are given
- Correct use of exit codes on success/failure of the script
- Presenting accurate and useful help if user makes an error
  - error messages must appear on standard error, not on standard output
  - error text must contain the name of the program issuing the error
  - always echo the input that is in error back to the user and give a hint as to the valid input that was expected, e.g.: echo 1>&2 "0: *Age*age is not between *min*$_{a}$*geand*max_age"

These criteria will be assessed in marking your programs (shell scripts).

2 Shell Script Structure

All major shell scripts in this course must have the following structure, laid out in the following 11-part order in the script file:

3. Syntax: how to use the script, using “man page” argument style
4. Purpose: longer description of how the script works, what it does, PDL
5. Course Assignment Label (7-8 lines)
6. Set PATH and export it
7. Set LC_COLLATE “C” locale collating sequence and export it
8. Set LANG “C” character set (one-byte characters only) and export it
9. Set umask (usually 022 to be friendly to others)
10. Do program algorithm (often four parts: Input, Validate, Process, Output)
11. Set exit status for script, and exit

An example of this format follows. My comments follow the example:

```
 1  #!/bin/sh -u
 2  # This shell script displays the command line arguments.
 3  # ----------------------------------------------------------------
 4  # Syntax:
 5  #   $0 [args...]
 6  # ----------------------------------------------------------------
 7  # Purpose: Count and display each argument to this shell script.
 8  #    It also displays the command name, which is often referred to as
 9  #    "Argument Zero" ($0) but is never counted as a command argument.
10  # ----------------------------------------------------------------
11  # Student Name:                 Ian! D. Allen
12  # Algonquin EMail Address:      alleni
13  # Student Number:               000-000-000
14  # Course Number:                CST8129
15  # Lab Section Number:           099
16  # Professor Name:               Dennis Ritchie & Brian Kernighan
17  # Assignment Name/Number/Date:  Exercise 123, due Sept 1, 2010
18  # Comment:                      This is a sample assignment label.
19  # ----------------------------------------------------------------
20
21  # Set the search path for the shell to be the standard places.
22  # Set the character collating sequence to be numeric ASCII/C standard.
23  # Set the character set to be standard one-byte ASCII.
24  # Set the umask to be non-restrictive and friendly to others.
25  #
26  PATH=/bin:/usr/bin ; export PATH
27  LC_COLLATE=C ; export LC_COLLATE
28  LANG=C ; export LANG
29  umask 022
30
31  # The $0 variable contains the name of this script, sometimes called
32  # "argument zero".  It is not an argument to the script itself.
33  # The script name is not counted as an "argument" to the script;
34  # only arguments 1 and up are counted as arguments to the script.
35  #
36  echo "Argument 0 is [$0]"
37
38  # Now display all of the command line arguments (if any).
39  # These are the real command line arguments to this script.
40  #
41  let count=0
42  for arg do
43          let count=count+1
44          echo "Argument $count is [$arg]"
45  done
46
47  exit 0
```

- **Line 1 [Section 1]** - A kernel binary program selection line
  - specify the correct program to interpret this script; spelling counts
  - this must be the very first line of the script (no blank lines allowed)
  - the `-u` option to Bourne shells checks for undefined variables
- **Line 2 [Section 2]** - Description: one-line description (man page title line)
  - *one line* explanation to summarize what the script does (one line!)
- **Lines 4-5 [Section 3]** - Syntax: how to use the script, in man page style
  - the syntax used to invoke the script on the command line
  - indicate which arguments are optional [] and what is expected for each
  - use "$0" to represent the script name in the syntax description; do not put the name of the script here (since it may change)
- **Lines 7-9 [Section 4]** - Purpose: description of how the script works, what it does
  - a paragraph or two explaining what the script does; may include PDL.
- **Lines 11-18 [Section 5]** - Course Assignment Label
  - the standard Assignment Submission Label, spelled correctly
- Lines 21-25, 31-35, 38-40 - block comments explaining the code that follows the block
- **Line 26 [Section 6]** - Set and export the search PATH for the commands found in this script
- **Line 27 [Section 7]** - Set the LC_COLLATE locale so that GLOB patterns and sorting work consistently (otherwise [a-z] may match upper-case letters!)
- **Line 28 [Section 8]** - Set the LANG character set so that programs expect single-byte chars
- **Line 29 [Section 9]** - Set the umask appropriately for this script (and its children).
  - note that PATH is a variable and umask is a *command*, not a variable
- **Lines 31-46 [Section 10]** - The script body goes here, using block-comment format. Often: Input, Validation, Process, Output
- **Line 47 [Section 11]** - Set the appropriate exit/return status

The above sections are explained more fully below.

2.1 The "shebang" interpreter line: #!/bin/sh -u

The first line (no leading spaces or blank lines allowed) of your script file must start with `#!` followed by the absolute pathname of the binary program that will be used to read the script file, followed by shell options. This line is almost always: `#!/bin/sh -u`

To catch errors (especially spelling errors) in your scripts, always specify the `-u` option to Bourne shell scripts. This option causes the script to fail and exit if you try to use the value of a variable that isn't defined. Without it, undefined variables (e.g. variable names that you spelled incorrectly) quietly expand to empty strings and your script misbehaves. Always check for undefined variables!

2.2 Internal Documentation

Internal documentation is documentation placed in the same file as the actual code you are writing, in the form of comments. You must have internal documentation in your major shell scripts.

to the ends of command lines; but, doing so makes your code harder to edit and maintain.

Lines in shell scripts should be kept to 80 characters or less. You can escape a newline with a backslash and continue on the next line, e.g.

```
echo "This is a very long line of script programming that needs" \
     "to be split into two parts to keep it under 80 columns."
```

Don't put the name of the shell script inside the shell script, since the name will be wrong if you rename the script. Use $0 instead, even in comments.

2.3 "Man" Page Syntax

The "man page style" syntax uses {} to surround mandatory items and [] to surround optional items. A program with one mandatory item that is either a file or a directory and one optional item that is one or more userids would have a Syntax line:

```
$0  {file|directory} [userid...]
```

Multiple optional arguments must nest:

```
$0 arg [ opt1 [ opt2 [ opt3 ] ] ]
```

which means that if you want to specify opt3 you have to specify opt1 and opt2 before it.

2.4 Script Header Lines

You must set the search PATH, umask value, locale, and language in your scripts; because, you have no idea what strange values might be inherited from the environment of the person running your script:

2.4.1 PATH

Choose PATH to include the system directories that contain the commands your script needs. Directories /bin and /usr/bin are almost always necessary. System scripts may need /sbin and /usr/sbin. GUI programs will need the X11 directories. Choose appropriately.

2.4.2 umask

Choose the umask to permit or restrict access to files and directories created by the shell script. "022" is a customary value, allowing read and execute by group and others. "077" is used in high-security scripts; since, it blocks all permissions for group and others.

You should set the umask even if your script does not itself create any new files or directories. Commands that you call may have hidden files created; or, you may later add commands that create files to your script.

2.4.3 LC_COLLATE

Set the character sort locale LC_COLLATE=C so that upper case letters sort before

a A b B c C .... x X y Y z Z

and so the GLOB pattern [a-z] (which we expect to mach only lower-case letters) actually matches a,A,b,B,…,x,X,y,Y,z and not 'Z'! You must set and export the "C" locale to fix this.

2.4.4 LANG

Set your character set to be "C" using LANG=C so that programs don't attempt to process multi-byte characters (UTF or UNICODE). If you are sure you want multi-byte processing, set LANG to the correct character set.

Be safe - always set PATH, umask, locale, and language.

2.5 Testing Return Codes

You must test the return codes of important commands inside the script. Your script must exit non-zero if an important command inside the script fails. (Read more on how to do this, below.)

2.6 Prompting on Standard Error

Prompts and error messages must be sent to Standard Error, not to Standard Output.

2.7 Quote All Variables

All non-numeric variables must be quoted correctly to prevent unexpected special character expansion by the shell. THIS IS IMPORTANT! DO IT!

2.8 Choose the correct Exit value

You will need to choose an appropriate exit value for your script. (Zero means "success", non-zero means "something went wrong".)

2.9 Script Template

If you prepare a template file containing the above script model, you can copy and use it to begin your scripts in labs and assignments and save time (and to avoid errors and omissions). Do not include my remarks.

3 Typical Four-Part Algorithm Structure

Functionally, shell scripts (like most Unix commands) typically have this form:

1. Input Stage

This Term Is Finished - Material may not be updated
- See current term for updates

    - complain about missing or extra arguments and exit non-zero
2. Validation Stage
    - validate the input: make sure numbers are numbers, files are files, directories are directories, etc.
    - test to make sure arguments are not empty or null strings
    - test to make sure things exist and are accessible
    - test to make sure things are readable/writable
3. Processing Stage
    - run your algorithm on your validated input
    - solve the problem; generate the results
4. Output Stage
    - display the output

Don't mix up code among the four stages. Keep them distinct:

- Do not duplicate your Processing Stage or Validation Stage for various combinations of Input arguments. Where possible, collect all the input first; then Validate it *once*, then write your Processing code *once*.
- Do not duplicate your Output Stage for various combinations of Input arguments. Generate your results and display the results *once*, at the end of the program. Where possible, don't duplicate similar kinds of output statements all over your code. Write one output statement that uses variables to contain the variable part of the output.

Less code is better code.

4 Prompts and Error Messages

Issue prompts on standard error before reading any input from the user; otherwise, the user won't know what to enter, or when. The prompt should explain exactly what kind of input is expected; don't just say "Enter input". Menus are also considered to be part of the prompts, and must also appear on standard error.

Always issue prompts and error messages on stderr (not on stdout), so that the prompts and error messages don't get redirected into output files:

```
echo 1>&2 "Enter student age:"
read student_age || exit 1
echo "You entered: $student_age"
```

After reading input, it is often a good idea to echo the input back to the user on standard output, so that they know what was entered. This is called "echoing the input back to the user", and is often a requirement in scripts submitted for marking.

Optional: Your script doesn't need to prompt the user to enter input if standard input is detected to be coming from something that is not a terminal keyboard (e.g. it might be

This Term Is Finished - Material may not be updated
- See current term for updates

4.1 Error Messages

Error messages must obey these four rules:

1. error messages must appear on standard error, not standard output
2. give the name of the program that is issuing the message (from $0)
3. state what input was expected (e.g. “expecting one file name”)
4. display what the user actually entered (e.g. “found 3 (a b c)”

Never say just “illegal input” or “invalid input” or “too many”. Always specify how many is “too many” or “too few”:

```
echo 1>&2 "$0: Expecting 3 file names; found $# ($*)"
echo 1>&2 "$0: Student age $student_age is not between" \
    "$min_age and $max_age"
echo 1>&2 "$0: Modify days $moddays is less than zero"
```

After detecting an error, the usual thing to do is to exit the script with a non-zero return code. Don’t keep processing bad data!

5 Commenting Shell Programs

Comments should add to a programmer’s understanding of the code. They don’t comment on the syntax or language mechanism used in the code; since, both these things are obvious to programmers who know the language. (Don’t comment that which is obvious to anyone who knows the language.)

“Programmer” comments deal with what the line of code means in the *algorithm* used (the “why”), not with syntax or how the language *works*.

“Teacher” or “Instructor” comments talk about how the *language* works, not about what the code means in the algorithm. Do not include Instructor comments in your code - I already know what the language means.

Thus: Do not use comments that state things that relate to the syntax or language mechanism used and are obvious to a programmer, e.g.

```
# THESE COMMENTS BELOW ARE OBVIOUS AND NOT HELPFUL COMMENTS:
#
x=$#                # set x to $#           <== OBVIOUS; NOT HELPFUL
date >x             # put date in x         <== OBVIOUS; NOT HELPFUL
test "$a" = "$b"    # see if $a equals $b   <== OBVIOUS; NOT HELPFUL
cp /dev/null x      # copy /dev/null to x   <== OBVIOUS; NOT HELPFUL
```

Better, programmer-style comments:

```
loop=$#                 # initialize loop index to max num arguments
```


This Term Is Finished - Material may not be updated
- See current term for updates

Do not copy “instructor-style” comments into your code. Instructor-style comments are put on lines of code by teachers to explain the language and syntax functions to people unfamiliar with programming (e.g. to students of the language). Instructor-style comments are “obvious” comments to anyone who knows how to program; they should never appear in your own programs (unless you become an instructor!).

6 Code Commenting Style

Comments should be grouped in blocks, ahead of blocks of related code to which the comments apply, e.g.

```
# Set standard PATH and secure umask for accounting file output.
#
PATH=/bin:/usr/bin ; export PATH
umask 077

# Verify that arguments exist and are non-empty.
#
NARGS=3
if test "$#" -ne $NARGS ; then
    echo 1>&2 "$0: Expecting $NARGS arguments; you gave: $# ($*)"
    exit 1
fi
for arg do
    if ! test -s "$arg" -a -f "$arg" ; then
        echo 1>&2 "$0: arg '$arg' is missing, empty, or a directory"
        exit 1
    fi
done
```

Do not alternate comments and single lines of code! This makes the code hard to read:

```
# BELOW IS A BAD BAD BAD EXAMPLE OF COMMENTS MIXED WITH CODE !!
#
# Set a standard PATH plus system admin directories
PATH=/bin:/usr/bin:/sbin:/usr/sbin
# export the PATH for other programs
export PATH
# Secure umask protects accounting files created by this script.
umask 077
# create empty lock file
>lockfile || exit $?
# attempt to create link to lock file
ln lockfile lockfile.tmp || exit $?
# copy password file in case of error
cp -p /etc/passwd /tmp/savepasswd$$ || exit $?
# remove guest account (can't quick-check return code on grep)
grep -v '^guest:' /etc/passwd >lockfile.tmp
# copy new file back to password file file system
cp lockfile.tmp /etc/passwd.tmp || exit $?
# fix the mode to be readable
chmod 444 /etc/passwd.tmp || exit $?
# use mv to do atomic update of passwd file
```


This Term Is Finished - Material may not be updated
- See current term for updates

```
rm lockfile.tmp lockfile
#
# ABOVE IS A BAD BAD BAD EXAMPLE OF COMMENTS MIXED WITH CODE !!
```

Block comments and code are easier to read. Here is a block-comment version of the above code:

```
# Set and export standard PATH plus system admin directories.
# Secure umask protects accounting files created by this script.
#
PATH=/bin:/usr/bin:/sbin:/usr/sbin ; export PATH
umask 077

# Create empty lock file and attempt to create link to lock file.
#
>lockfile || exit $?
ln lockfile lockfile.tmp || exit $?

# Copy password file in case of error.
#
cp -p /etc/passwd /tmp/savepasswd$$ || exit $?

# Remove guest account into temp file.
# (Note: You can't quick-check the return code on grep using || or &&.)
# Copy the temp file back to password file file system.
# Fix the mode to be readable.
# Use mv to do atomic update of passwd file.
# Remove the temp file and the lock file.
#
grep -v '^guest:' /etc/passwd >lockfile.tmp
cp lockfile.tmp /etc/passwd.tmp || exit $?
chmod 444 /etc/passwd.tmp || exit $?
mv /etc/passwd.tmp /etc/passwd || exit $?
rm lockfile.tmp lockfile
```

Question: Why can't you exit the script if grep returns a non-zero status code?

7 Writing and Debugging Shell Scripts

The **Number One** rule of writing shell scripts is:

> *Start Small and Add One Line at a Time!*

Students who write a 10- or 100-line script and then try to test it all at once usually run out of time. An unmatched quote at the start of a script can eat the entire script until the next matching quote!

If you disobey this Number One rule and add many lines to a script and then find it's broken due to some shell syntax error, your only option is to start deleting lines and re-running the

Start your script with the Script Header (name of interpreter, PATH, umask, comments) and the single command "date". If that doesn't work, you know something fundamental is wrong, and you only have a few lines of code that you need to debug. (Is your interpreter correct? your PATH?)

Add to this simple script one or two lines at a time, so that when an error occurs you know it must be in the last line or two that you added.

Do not add 10 lines to a script! You won't know what you did wrong!

You can ask the a shell to show you the lines of the script it is reading and executing by using the "-v" or "-x" (or both) option to the shell:

```
$ sh -v -u ./myscript arg1 arg2 ...
$ sh -x -u ./myscript arg1 arg2 ...
```

The "-v" options displays all lines (including comment lines) as they are read by the shell (without any shell expansion). The "-x" option displays only the command lines as they are passed to the commands being executed, after the shell has done all the command line expansion and processing.

These options will allow you to see the commands as they execute, and may help you locate errors in your script. (Double-quote your variables!)

Of course you can use -v and -x with an interactive shell too:

```
$ sh -v
$ echo $SHELL
echo $SHELL
/usr/bin/ksh
$ echo *
echo *
a b c d

$ sh -x
$ echo $SHELL
+ echo /usr/bin/ksh
/usr/bin/ksh
$ echo *
+ echo a b c d
a b c d

$ sh -v -x
$ echo $SHELL
echo $SHELL
+ echo /usr/bin/ksh
/usr/bin/ksh
$ echo *
echo *
+ echo a b c d
a b c d
```

Remember that if you use a shell to read a shell script ("sh scriptname"), instead of executing it directly ("./scriptname"), the shell will treat all the comments at the start of the shell script as comments. In particular, the comment that specifies the interpreter to use when executing the script ("#!/bin/sh -u") will be ignored, as will all of the options listed

Only by actually *executing* the script will you cause the Unix kernel to use the interpreter and options given on the first line of the script. If you just pass the script to a shell, none of those options apply. For example:

```
$ cat test
#!/bin/sh -u
echo 1>&2 "$0: This is '$undefined'"

$ ./test
./test: undefined: unbound variable

$ sh test
test: This is ''

$ sh -u test
test: undefined: unbound variable

$ csh test
Bad : modifier in $ ( ).
```

All shells treat #-lines as comments and ignore them. Only the Unix kernel treats #! specially, and only for executable scripts. The #! line must be the very first line of the script; no blank lines allowed.

8 Complementing Return Status

Any command or command pipeline's return status can be complemented (reversed from good to bad or bad to good) using a leading "!" before the command, e.g.

```
$ false
$ echo $?
1

$ ! false
$ echo $?
0

$ grep nosuchxxx /etc/passwd
$ echo $?
1

$ ! grep nosuchxxx /etc/passwd
$ echo $?
0
```

This is useful in shell scripts to simplify this:

```
if grep "$var" /etc/passwd ; then
    : do nothing
else
    echo 1>&2 "$0: Cannot find '$var' in /etc/passwd; status $?"
fi
```


This Term Is Finished - Material may not be updated
- See current term for updates

```
if ! grep "$var" /etc/passwd ; then
    echo 1>&2 "$0: Cannot find '$var' in /etc/passwd"
fi
```

Note that if you use "!" to turn a bad status into a good one, you cannot echo the failing non-zero status value in your error message; because, the "!" has changed the $? status from non-zero to zero:

```
if ! grep nosuchstring /etc/passwd ; then
    echo 1>&2 "$0: grep failed, status is $?"
fi
```

The above code always prints "status is 0" because the "!" changes the failing non-zero exit status of grep into zero (so that the IF succeeds), and it is that successful zero that is put into $? and echoed. If you want to know the failing exit status, you cannot use a leading "!":

```
if grep nosuchstring /etc/passwd ; then
    : do nothing
else
    echo 1>&2 "$0: grep failed, status is $?"
fi
```

The above code prints the non-zero exit status of grep correctly.

The "!" prefix is also useful in turning "while" loops into "until" loops or vice-versa.

9 Testing Return Codes of Commands

Just as you would never use a C Library function without checking its return code, you must never use commands in important shell scripts without at least a minimal checking of their return codes. At minimum, the shell script should exit non-zero if a command fails unexpectedly:

```
grep -v '^guest:' /etc/passwd >lockfile.tmp
cp lockfile.tmp /etc/passwd.tmp                 || exit $?
chmod 444 /etc/passwd.tmp                       || exit $?
mv /etc/passwd.tmp /etc/passwd                  || exit $?
rm lockfile.tmp lockfile                        || exit $?
```

The shell conditional execution syntax "||" is used here to test the return codes of the commands on the left and execute the command on the right if the command on the left returns a bad status (non-zero).

Some commands naturally return a non-zero exit status even when they are doing what you expect (e.g. grep might not find what you were looking for - this might be okay), and cannot be tested using this simple method. Do not exit the script after a "grep", "diff", or "cmp" command retuns non-zero, since these commands do sometimes set a non-zero return code!

This Term Is Finished - Material may not be updated
- See current term for updates

Unfortunately, simply exiting non-zero doesn't tell the user of the script which script contained the command that failed:

```
$ ./myscript1 &                         <== start in background
$ ./myscript2 &                         <== start in background
$ ./myscript3 &                         <== start in background
[...time passes...]
cp: /etc/passwd.tmp: No space left on device
```

From which script did the error message come? We don't know!

If a script is being run in the background along with several other scripts also containing similar commands, or if a script is being run by a system daemon or delayed execution scheduler (atd or crond), we won't know from which script the actual "cp" error message came.

More work is needed to produce a truly useful error message when a command inside a script fails.

The full and proper way to handle non-zero return codes in scripts is by using error messages that contain the script name. This means you need "if" statements around *every* command that might fail! This is probably overkill for most hobby scripts; but, it is necessary for robust systems programming:

```
grep -v '^guest:' /etc/passwd >lockfile.tmp
status="$?"
if [ "$status" -ne 1 -a "$status" -ne 0 ] ; then
    # grep returns 2 on serious error
    echo 1>&2 "$0: grep guest /etc/passwd failed; status $status"
    exit 1
fi
if ! cp lockfile.tmp /etc/passwd.tmp ; then
    echo 1>&2 "$0: cp lockfile.tmp /etc/passwd.tmp failed"
    exit 1
fi
if ! chmod 444 /etc/passwd.tmp ; then
    echo 1>&2 "$0: chmod /etc/passwd.tmp failed"
    exit 1
fi
if ! mv /etc/passwd.tmp /etc/passwd ; then
    echo 1>&2 "$0: mv /etc/passwd.tmp /etc/passwd failed"
    exit 1
fi
rm lockfile.tmp lockfile
```

The above script now tells you its name in the error message:

```
$ ./myscript
cp: /etc/passwd.tmp: No space left on device
./myscript: cp lockfile.tmp /etc/passwd.tmp failed
```

Now it's easy to tell from which script the above "cp" error came.

Making your scripts detect errors and issue clear error messages is tedious but not difficult. Adding all the error checking makes the code much longer and harder to read and modify. If the script isn't doing anything important, simply exiting after a failed command (using the

For a system script that must detect errors under all conditions (including “too many processes”, “file system full”, etc.), you must have all the additional error checking. The reward is a script that won’t let you down when things go wrong, and that will tell you exactly what the problem is when one develops.

For scripts in this course, using the quick-exit “||” syntax to test the return code of major commands is usually sufficient. What is a “major” command? Something that, if it failed, would make the rest of the script misbehave in a serious way. Issuing status messages via “echo” is usually not a “major” command. Changing directories or copying files is usually “major”. Here is a short example of a script that must use “||”:

```
cd "$DIR" || exit $?
rm *
```

If the “cd $DIR" fails, the subsequent "rm *" will remove all the files in the *current* directory, not the $DIR directory. You must ensure that the script exits if the”cd" fails!

10 Naming Shell Scripts - $0

Often, you will want to put an example of how to run a shell script inside the shell script as a comment. You might create a script called “doexec” and write a comment in it as follows:

```
#!/bin/sh -u
# This script sets execute permissions on all its arguments.
#     doexec [ files... ]      (* don't do it this way; see below!*)
# -IAN! idallen@idallen.ca   Mon Jun 11 23:02:38 EDT 2001

PATH=/bin:/usr/bin ; export PATH
umask 022

if [ "$#" -eq 0 ]; then
    echo 1>&2 "$0: No arguments; nothing done"
    status=0
else
    if chmod +x "$@" ; then
        status=0   # it worked
    else
        status="$?"
        echo 1>&2 "$0: chmod exit status: $status"
        echo 1>&2 "$0: Could not change mode of some argument: $*"
    fi
fi
exit "$status"
```

You would execute this script by typing:

```
$ ./doexec filename1 filename2 filename3...
```

This comment line in the doexec script:

This Term Is Finished - Material may not be updated
- See current term for updates

tells the reader that the script name is “doexec” and the files are the (optional) arguments to the script. This is the “syntax” of the command.

But what if you rename the script to be something other than “doexec”?

```
$ mv doexec fixperm
$ ./fixperm foo bar
```

Now the comment is wrong. The script is named “fixperm”, not “doexec”.

The use of “$0” in the echo line for the error message ensures that the shell will print the actual script name in the error message, but the comment in the script is now wrong, since the program name is no longer “doexec”. I don’t want to have to edit the script and make a change such as “doexec” to “fixperm” every time I change the name of the script.

The solution is never to put the actual name of a script inside the script, even as a comment. Wherever you refer to the name of the script, even in a comment, use the “$0” convention instead. So, the comment changes from:

```
#     doexec [ files... ]
```

to be:

```
#     $0 [ files... ]
```

In the comment, “$0” just means “whatever the name of this script is”, without my having to actually write the script name. I don’t want to use the actual script name, because I might change it. Since the line is a comment, ignored by the shell, the shell will never actually expand that “$0” to be the real name of the shell; it’s just a convenient way of specifying a place holder for the program name without actually naming it inside the script.

Never put the name of a program inside the program; it might change!

THIS TERM IS FINISHED - MATERIAL MAY NOT BE UPDATED
- SEE CURRENT TERM FOR UPDATES

Author:
| Ian! D. Allen - idallen@idallen.ca - Ottawa, Ontario, Canada
| Home Page: http://idallen.com/ Contact Improv: http://contactimprov.ca/
| College professor (Free/Libre GNU+Linux) at: http://teaching.idallen.com/
| Defend digital freedom: http://eff.org/ and have fun: http://fools.ca/

Plain Text - plain text version of this page in Pandoc Markdown format

---

 Author Ian! D. Allen

THIS TERM IS FINISHED | CONTENT MAY NOT BE UPDATE

This Term Is Finished - Material may not be updated
- See current term for updates