Debugging a script

These few lines are not intended as a full-fledged debugging tutorial, but as hints and comments about debugging a Bash script.

Use a unique name for your script

Do **not** name your script `test`, for example! *Why?* `test` is the name of a UNIX®-command, and most likely built into your shell (it's a built-in in Bash) - so you won't be able to run a script with the name `test` in a normal way.

Don't laugh! This is a classic mistake

Read the error messages

Many people come into IRC () and ask something like *"Why does my script fail? I get an error!"*. And when you ask them what the error message is, they don't even know. Beautiful.

Reading and interpreting error messages is 50% of your job as debugger! Error messages actually **mean** something. At the very least, they can give you hints as to where to start debugging. **READ YOUR ERROR MESSAGES!**

You may ask yourself why is this mentioned as debugging tip? Well, you would be surprised how many shell users ignore the text of error messages! When I find some time, I'll paste 2 or 3 IRC () log-snips here, just to show you that annoying fact.

Use a good editor

Your choice of editor is a matter of personal preference, but one with **Bash syntax highlighting** is highly recommended! Syntax highlighting helps you see (you guessed it) syntax errors, such as unclosed quotes and braces, typos, etc.

From my personal experience, I can suggest `vim` or `GNU emacs`.

Write logfiles

For more complex scripts, it's useful to write to a log file, or to the system log. Nobody can debug your script without knowing what actually happened and what went wrong.

An available syslog interface is `logger` (online manpage (http://unixhelp.ed.ac.uk/CGI/man-cgi?logger+1)).

Inject debugging code

Insert **echos** everywhere you can, and print to `stderr` :

```
echo "DEBUG: current i=$i" >&2
```

If you read input from **anywhere**, such as a file or command substitution, print the debug output with literal quotes, to see leading and trailing spaces!

```
pid=$(< fooservice.pid)
echo "DEBUG: read from file: pid=\"$pid\"" >&2
```

Bash's printf command has the `%q` format, which is handy for verifying whether strings are what they appear to be.

```
foo=$(< inputfile)
printf "DEBUG: foo is |%q|\n" "$foo" >&2
# exposes whitespace (such as CRs, see below) and non-printing charac
ters
```

Use shell debug output

There are two useful debug outputs for that task (both are written to `stderr`):

- `set -v` mode (`set -o verbose`)
  - print commands to be executed to `stderr` as if they were read from input (script file or keyboard)
  - print everything **before** any (substitution and expansion, …) is applied
- `set -x` mode (`set -o xtrace`)
  - print everything as if it were executed, after substitution and expansion is applied
  - indicate the depth-level of the subshell (by default by prefixing a `+` (plus) sign to the displayed command)
  - indicate the recognized words after word splitting by marking them like `'x y'`
  - in shell version 4.1, this debug output can be printed to a configurable file descriptor, rather than sdtout by setting the BASH_XTRACEFD variable.

Hint: These modes can be entered when calling Bash:

- from commandline: `bash -vx ./myscript`
- from shebang (OS () dependant): `#!/bin/bash -vx`

Simple example of how to interpret xtrace output

Here's a simple command (a string comparison using the classic test command) executed while in `set -x` mode:

```
set -x
foo="bar baz"
[ $foo = test ]
```

That fails. Why? Let's see the `xtrace` output:

```
+ '[' bar baz = test ']'
```

And now you see that it's ("bar" and "baz") recognized as two separate words (which you would have realized if you READ THE ERROR MESSAGES ;)). Let's check it…

```
# next try
[ "$foo" = test ]
```

`xtrace` now gives

```
+ '[' 'bar baz' = test ']'
      ^       ^
    word markers!
```

Making xtrace more useful

(by AnMaster)

`xtrace` output would be more useful if it contained source file and line number. Add this assignment PS4 at the beginning of your script to enable the inclusion of that information:

```
export PS4='+(${BASH_SOURCE}:${LINENO}): ${FUNCNAME[0]:+${FUNCNAME
[0]}(): }'
```

Be sure to use single quotes here!

The output would look like this when you trace code *outside a function*:

```
+(somefile.bash:412): echo 'Hello world'
```

…and like this when you trace code *inside a function*:

```
+(somefile.bash:412): myfunc(): echo 'Hello world'
```

That helps a lot when the script is long, or when the main script sources many other files.

Set flag variables with descriptive words

If you test variables that flag the state of options, such as with `if [[ -n $option ]];`, consider using descriptive words rather than short codes, such as 0, 1, Y, N, because xtrace will show `[[ -n word ]]` rather than `[[ -n 1 ]]` when the option is set.

Debugging commands depending on a set variable

For general debugging purposes you can also define a function and a variable to use:

```
debugme() {
 [[ $script_debug = 1 ]] && "$@" || :
 # be sure to append || : or || true here or use return 0, since the
return code
 # of this function should always be 0 to not influence anything else
with an unwanted
 # "false" return code (for example the script's exit code if this fu
nction is used
 # as the very last command in the script)
}
```

This function does nothing when `script_debug` is unset or empty, but it executes the given parameters as commands when `script_debug` is set. Use it like this:

```
script_debug=1
# to turn it off, set script_debug=0

debugme logger "Sorting the database"
database_sort
debugme logger "Finished sorting the database, exit code $?"
```

Of course this can be used to execute something other than echo during debugging:

```
debugme set -x
# ... some code ...
debugme set +x
```

Dry-run STDIN driven commands

Imagine you have a script that runs FTP () commands using the standard FTP () client:

```
ftp user@host <<FTP
cd /data
get current.log
dele current.log
FTP
```

A method to dry-run this with debug output is:

```
if [[ $DRY_RUN = yes ]]; then
  sed 's/^/DRY_RUN FTP: /'
else
  ftp user@host
fi <<FTP
cd /data
get current.log
dele current.log
FTP
```

This can be wrapped in a shell function for more readable code.

Common error messages

Unexpected end of file

```
script.sh: line 100: syntax error: unexpected end of file
```

Usually indicates exactly what it says: An unexpected end of file. It's unexpected because Bash waits for the closing of a compound command:

- did you close your `do` with a `done`?
- did you close your `if` with a `fi`?
- did you close your `case` with a `esac`?
- did you close your `{` with a `}`?
- did you close your `(` with a `)`?

Note: It seems that here-documents (tested on versions `1.14.7`, `2.05b`, `3.1.17` and `4.0`) are correctly terminated when there is an EOF () before the end-of-here-document tag (see redirection). The reason is unknown, but it seems to be deliberate. Bash 4.0 added an extra message for this: `warning: here-document at line <N> delimited by end-of-file (wanted `<MARKER>')`

Unexpected end of file while looking for matching ...

```
script.sh: line 50: unexpected EOF while looking for matching `"'
script.sh: line 100: syntax error: unexpected end of file
```

This one indicates the double-quote opened in line 50 does not have a matching closing quote.

These *unmatched errors* occur with:

- double-quote pairs
- single-quote pairs (also `$'string'`!)
- missing a closing `}` with parameter expansion syntax

Too many arguments

```
bash: test: too many arguments
```

You most likely forgot to quote a variable expansion somewhere. See the example for `xtrace` output from above. External commands may display such an error message though in our example, it was the **internal** test-command that yielded the error.

!": event not found

```
$ echo "Hello world!"
bash: !": event not found
```

This is not an error per se. It happens in interactive shells, when the C-Shell-styled history expansion (" `!searchword` ") is enabled. This is the default. Disable it like this:

```
set +H
# or
set +o histexpand
```

syntax error near unexpected token `('

When this happens during a script **function definition** or on the commandline, e.g.

```
$ foo () { echo "Hello world"; }
bash: syntax error near unexpected token `('
```

you most likely have an alias defined with the same name as the function (here: `foo`). Alias expansion happens before the real language interpretion, thus the alias is expanded and makes your function definition invalid.

The CRLF issue

What is the CRLF issue?

There's a big difference in the way that UNIX® and Microsoft® (and possibly others) handle the **line endings** of plain text files. The difference lies in the use of the CR (Carriage Return) and LF (Line Feed) characters.

- MSDOS uses: `\r\n` (ASCII () `CR` #13 `^M` , ASCII () LF #10)
- UNIX® uses: `\n` (ASCII () `LF` #10)

Keep in mind your script is a **plain text file**, and the `CR` character means nothing special to UNIX® - it is treated like any other character. If it's printed to your terminal, a carriage return will effectively place the cursor at the beginning of the *current* line. This can cause

much confusion and many headaches, since lines containing CRs are not what they appear to be when printed. In summary, CRs are a pain.

How did a CR end up in my file?

Some possible sources of CRs:

- a DOS/Windows text editor
- a UNIX® text editor that is "too smart" when determining the file content type (and thinks "*it's a DOS text file*")
- a direct copy and paste from certain webpages (some pastebins are known for this)

Why do CRs hurt?

CRs can be a nuisance in various ways. They are especially bad when present in the shebang/interpreter specified with `#!` in the very first line of a script. Consider the following script, written with a Windows® text editor (`^M` is a symbolic representation of the `CR` carriage return character!):

```
#!/bin/bash^M
^M
echo "Hello world"^M
...
```

Here's what happens because of the `#!/bin/bash^M` in our shebang:

- the file `/bin/bash^M` doesn't exist (hopefully)
- So Bash prints an error message which (depending on the terminal, the Bash version, or custom patches!) may or may not expose the problem.
- the script can't be executed

The error message can vary. If you're lucky, you'll get:

```
bash: ./testing.sh: /bin/bash^M: bad interpreter: No such file or dir
ectory
```

which alerts you to the CR. But you may also get the following:

```
: bad interpreter: No such file or directory
```

Why? Because when printed literally, the `^M` makes the cursor go back to the beginning of the line. The whole error message is *printed*, but you *see* only part of it!

It's easy to imagine the `^M` is bad in other places too. If you get weird and illogical messages from your script, rule out the possibility that `^M` is involved. Find and eliminate it!

How can I find and eliminate them?

To display CRs (these are only a few examples)

- in VI/VIM: `:set list`

- with `cat(1)`: `cat -v FILE`

To eliminate them (only a few examples)

- blindly with `tr(1)`: `tr -d '\r' <FILE >FILE.new`
- controlled with `recode(1)`: `recode MSDOS..latin1 FILE`
- controlled with `dos2unix(1)`: `dos2unix FILE`

See also

- the set builtin command (for `-v` and `-x`)

Fix Me!

- DEBUG trap
- BASH Debugger http://bashdb.sourceforge.net/ (http://bashdb.sourceforge.net/)

Discussion

avkosinsky (http://sourceforge.net/projects/basheclipse/), 2011/10/18 13:20 ()

Debugger for Bash version 3(Bourne again shell). Plugin for Eclipse.
http://sourceforge.net/projects/basheclipse/
(http://sourceforge.net/projects/basheclipse/)

Paolo Supino, 2012/01/02 09:20 (), 2012/01/02 19:13 ()

not knowing of bash debugger existance I wrote a small script (I called debug.sh) that sets -x, -xv or -xvn (depending on the parameter passed debug.sh). The debug.sh script is (feel free to copy, use and evolve it as you see fit):

```
#!/bin/bash

PS4='+(${BASH_SOURCE}:${LINENO}): ${FUNCNAME[0]:+${FUNCNAME[0]}
(): }'

export PS4

usage()
{
cat <<'EOF'
usage: debug [option] script arguments
possible options are:

- help|usage: print this screen
- verbose: sets -xv flags
- noexec: sets -xvn flags
- no parameter sets -x flags
EOF

fmt << 'EOF'

 if the script takes arguments remember to enclose the script and
arugments
 in ""
EOF
}

debug_cmd()
{
   /bin/bash $FLAGS $SCRIPT
}

if [ $# -gt 0 ]; then
   case "$1" in
         "verbose")
            FLAGS=-xv
            SCRIPT=$2
         ;;
         "noexec")
            FLAGS=-xvn
            SCRIPT=$2
         ;;
         "help"|"usage")
            usage
            exit 3
         ;;
         *)
            FLAGS=-x
            SCRIPT=$1
         ;;
   esac
   debug_cmd
else
   usage
fi
```

Paolo Supino, 2012/06/22 09:12 (), 2012/06/22 17:51 ()

I've updated the debug.sh posted above to display colors in the ouput when using only +x (makes everything more readable). the new script is:

```
#!/bin/bash

color_def="~/.colorrc"

if [[ -f $color_def ]]; then
   . $color_def
else
   # color definitions
   black="$(tput setaf 0)"
   darkgrey="$(tput bold ; tput setaf 0)"
   lightgrey="$(tput setaf 7)"
   white="$(tput bold ; tput setaf 7)"
   red="$(tput setaf 1)"
   lightred="$(tput bold ; tput setaf 1)"
   green="$(tput setaf 2)"
   lightgreen="$(tput bold ; tput setaf 2)"
   yellow="$(tput setaf 3)"
   blue="$(tput setaf 4)"
   lightblue="$(tput bold ; tput setaf 4)"
   purple="$(tput setaf 5)"
   pink="$(tput bold ; tput setaf 5)"
   cyan="$(tput setaf 6)"
   lightcyan="$(tput bold ; tput setaf 6)"
   nc="$(tput sgr0)" # no color
fi
export darkgrey lightgreywhite red lightred green lightgreen yell
ow blue
export lightblue purple pink cyan lightcyan nc
if [[ ! $level_color ]]; then
   level_color=$cyan
fi
if [[ ! $script_color ]]; then
   script_color=$yellow
fi
if [[ ! $linenum_color ]]; then
   linenum_color=$red
fi
if [[ ! $funcname_color ]]; then
   funcname_color=$green
fi
if [[ ! $command_color ]]; then
   command_color=$white
fi
export script_color linenum_color funcname_color

reset_screen() {

   echo $nc
}
reset_screen

usage()
{
cat <<'EOF'
```

```
usage: debug [option] script arguments

possilbe options are:
- help|usage: print this screen
- verbose: sets -xv flags
- noexec: sets -xvn flags
- no parameter sets -x flags

EOF
fmt <<EOF
if the script takes arguments remember to enclose the script and
 arugments
in ""
EOF

fmt <<EOF

The script prints the script name, script line number and functio
n name as it
executes the script. The various parts of the script prompt are p
rinted in
color. If the default colors are not suitable than you can set th
e environment
varialbes script_color linenum_color funcname_color to any of the
following
colors: ${darkgrey}darkgrey$nc, ${lightgrey}light grey$nc, ${whit
e}white,
${red}red, ${lightred}light red, ${green}green, ${lightgreen}ligh
t green,
${yellow}yellow, ${blue}blue, ${lightblue}light blue, ${purple}pu
rple,
${pink}pink, ${cyan}cyan, ${lightcyan}light cyan$nc.
EOF

cat <<EOF

default colors are:
${level_color}- shell level color:cyan$nc
${script_color}- script name: yellow$nc
${linenum_color}- line number: red$nc
${funcname_color}- function name: green$nc
${command_color}- command executed: white'$nc
EOF
}


debug_cmd()
{
   trap reset_screen INT
   /bin/bash $FLAGS $SCRIPT
}

if [ $# -gt 0 ]; then
   case "$1" in
         "verbose")
            FLAGS=-xv
```

```
            SCRIPT=$2
         ;;
         "noexec")
            FLAGS=-xvn
            SCRIPT=$2
         ;;
         "help"|"usage")
            usage
            exit 3
         ;;
         *)
            FLAGS=-x
            PS4="${level_color}+${script_color}"'(${BASH_SOURCE##
*/}:${linenum_color}${LINENO}${script_color}):'" ${funcname_colo
r}"
            export PS4
            SCRIPT=$1
         ;;
   esac
   debug_cmd
else
   usage
fi

reset_screen
```

Paolo Supino, 2012/07/04 07:29 ()

My last debug.sh version had a couple of bugs: 1: it didn't print the function name when run with only -x flag. 2: The first line of prompt had a different coloring scheme than the rest of the lines… In the version below I fixed those problems. Please delete my previous version and post this one instead. Thanx.

#!/bin/bash

color_def="~/.colorrc"

if -f $color_def; then

```
 . $color_def
```

else

```
# color definitions
black="$(tput setaf 0)"
darkgrey="$(tput bold ; tput setaf 0)"
lightgrey="$(tput setaf 7)"
white="$(tput bold ; tput setaf 7)"
red="$(tput setaf 1)"
lightred="$(tput bold ; tput setaf 1)"
green="$(tput setaf 2)"
lightgreen="$(tput bold ; tput setaf 2)"
yellow="$(tput setaf 3)"
blue="$(tput setaf 4)"
lightblue="$(tput bold ; tput setaf 4)"
purple="$(tput setaf 5)"
pink="$(tput bold ; tput setaf 5)"
cyan="$(tput setaf 6)"
lightcyan="$(tput bold ; tput setaf 6)"
nc="$(tput sgr0)" # no color
```

fi export darkgrey lightgreywhite red lightred green lightgreen yellow blue export lightblue purple pink cyan lightcyan nc if ! $lc; then

```
lc=$cyan
```

fi if ! $sc; then

```
sc=$yellow
```

fi if ! $lnc; then

```
lnc=$red
```

fi if ! $fc; then

```
fc=$green
```

fi if ! $cc; then

```
cc=$white
```

fi export sc lnc fc

reset_screen() {

```
echo $nc
```

} reset_screen

usage() { cat «'EOF'

usage: debug [option] script arguments

possilbe options are: - help|usage: print this screen - test|compile: sets -n flag - verbose: sets -xv flags - noexec: sets -xvn flags - no parameter sets -x flag

EOF fmt «EOF if the script takes arguments remember to enclose the script and arugments in "" EOF

fmt «EOF

The script prints the script name, script line number and function name as it executes the script. The various parts of the script prompt are printed in color. If the default colors are not suitable than you can set the environment varialbes sc lnc fc to any of the following colors: ${darkgrey}darkgrey$nc, ${lightgrey}light grey$nc, ${white}white, ${red}red, ${lightred}light red, ${green}green, ${lightgreen}light green, ${yellow}yellow, ${blue}blue, ${lightblue}light blue, ${purple}purple, ${pink}pink, ${cyan}cyan, ${lightcyan}light cyan$nc. EOF

cat «EOF

default colors are: ${lc}- shell level color: cyan ${sc}- script name: yellow ${lnc}- line number: red ${fc}- function name: green ${cc}- command executed: white EOF }

debug_cmd() {

```
trap reset_screen INT
/bin/bash $FLAGS $SCRIPT
```

}

if [$# -gt 0]; then

```
case "$1" in
      "test"|"compile")
         FLAGS=-n
         SCRIPT=$2
      ;;
      "verbose")
         FLAGS=-xv
         SCRIPT=$2
      ;;
      "noexec")
         FLAGS=-xvn
         SCRIPT=$2
      ;;
      "help"|"usage")
         usage
         exit 3
      ;;
      *)
         FLAGS=-x
         PS4="${white}${lc}+${sc}"'(${BASH_SOURCE##*/}'":${ln
c}"'${LINENO}'"${sc}): ${fc}"'${FUNCNAME[0]}'"(): ${cc}"
         export PS4
         SCRIPT=$1
      ;;
esac
debug_cmd
```

else

```
usage
```

fi

reset_screen

Robert Wlaschin, 2012/09/27 06:08 ()

This is a great article. I have a suggestion for putting in DEBUG switches.

Putting a line like the following:

debug switch [-z "$DEBUG"] && DEBUG=0 || :

…

[$DEBUG = 0] || echo "Debug output"

Will allow passing in a value through environment variables, meaning that code does not need to be changed or put in unnecessary command-line arguments to enable debugging

This is how it could be used

turn on debugging DEBUG=1 ./runtest

regular no debug ./runtest

Thoughts?

scripting/debuggingtips.txt Last modified: 2017/06/07 02:42 by fgrose