CST8177

bash Scripting

Chapters 13 and 14 in Quigley's
"UNIX Shells by Example"

Signals and the TRAP statement

- Page 935 - 941 in Quigley.
- Various signals can be trapped and your own script code executed instead of the system's normal code. Although there are up to 64 signals available, we will consider only a few of them:
- **SIGHUP** (signal 1 or **HUP**: hang up) is issued for a remote connection when the connection is lost or terminated; it's also used to tap a daemon on the shoulder, to re-read its config files.
- **SIGINT** (signal 2 or **INT**) is the keyboard interrupt signal given by **Control-C**.
- **SIGKILL** (signal 9 or **KILL**) cannot be ignored or trapped.
- **SIGTERM** (signal 15 or **TERM**) is the default signal used by kill(1) and killall(1).

Signal-like events and TRAP

- The **EXIT** event (also "signal" 0) occurs upon exit from the current shell.
- The **DEBUG** event takes place before every simple command, **for** command, **case** command, **select** command, and before the first command in a function. See also the description of **extdebug** for the **shopt** built-in for details of its effect.
- The **ERR** event takes place for each simple command with a non-zero exit status, subject to these conditions: it is not executed if the failed command is part of a **while**, **until**, or **if** condition expression, or in a **&&** or **||** list, or if the command's return value is being inverted via **!**. See also **errexit** for details.
- The **RETURN** event occurs each time a shell function or a script executed with the **.** (that's a dot) or **source** built-in returns to its caller.

Signals and the TRAP statement

- You can set a trap:

  `trap 'statement; statement; …' event-list`

- The trap statement list is read by the shell twice, first when it's set (it's set once only, before it is to be used, and stays active until you clear it).
- It's read a second time when it's executed.
- If you enclose the statement in single quotes, substitutions only take place at execution time.
- If you use double quotes, substitutions will take place upon both readings.
- If statement is omitted, the signals (use **-** (dash)) for all) are reset to the default.
- If statement is a null (empty) string, the signals specified will be ignored.

Signals and the TRAP statement

- To set a trap for **SIGINT**:

  trap 'statement; statement; …' INT

- To turn it off again:

  trap INT

- To prevent any **SIGINT** handling (ignore signals):

  trap " " INT

- Be cautious in trapping **SIGINT**: how will you stop a run-away script?
- To see what traps are set (you can see traps for specific events by listing the names or numbers):

  trap -p

- To list the names for signals **1** to **SIGRTMAX**:

  trap -l   # that's an ell, not a one

Trap Sample Script

```
#! /bin/bash
declare -i count=0

# set trap to echo, then turn itself off
trap 'echo -e \\nSIGINT ignored in $count; \
     trap sigint;' sigint

# loop forever
while (( 1 == 1 )); do
    let count++
    read -p "$count loop again? "
done

# if loop ends, display count
echo loop count $count
exit 0
```

```
System Prompt$ ./my-trap
1 loop again?
2 loop again?
3 loop again?
4 loop again? y
5 loop again? n
6 loop again?
7 loop again? q
8 loop again? help
9 loop again? ^C
SIGINT ignored in 9

10 loop again? q
11 loop again? y
12 loop again? n
13 loop again? ^C
System Prompt$
```

Functions in bash

- You will learn that functions are exceptionally useful, and it's good to see them in bash.
- Pages 927 - 935 in Quigley
- A function is a group of regular shell-script statements is a self-contained package.
- You define a function as:

```
function somename ()
{
    statement
    statement
     …
}
```

- And you call it by using the name as if it were a normal command.

bash Functions

A function does not need to return a result, but it may do so in 3 ways (perhaps all in the same function):

1. set a new value into a variable previously defined outside the function;
2. Use a **`return`** statement to set the value of **`$?`**; it can also use an **`exit`** to set **`$?`**, but that will also exit from the calling script which may not be what you want.
3. write the results to **`stdout`**.

Functions

- Function scope is from the point the function is defined to the end of the file (that is, it must be defined before you can use it). Generally, that means that all functions precede the main body of the script.
- As a result, previously-written functions are often included in a script using the **`source`** (also **`.`** (dot)) statement near the top of a script.
- You can define **`local`** variables to be used only inside the function, while your normal variables from outside the function can always be used.
- If you wish, you can pass arguments into a function as positional parameters (**`$1`** and so on; this is by far the recommended approach).

Functions

- You may have noticed that traps behave like a special form of function. They are called (or invoked) by an event and consist of a collection of command statements. This is not an accident.
- To unset (delete or remove) a function:

  `unset -f functionname`
- To list defined function names (note: my system seems to have over 400 functions, of which I have only defined 4 of my own):

  `declare -F | less`
- To list functions and definitions:

  `declare -f  [functionname]`

A Simple Sample

The **rot13** script is an implementation of the Caesar code message encryption. It simply rotates the message 13 characters through the alphabet, retaining case. No numbers or punctuation characters are affected.

```
rot13 ()
{
    echo "$*" | \
         tr '[a-mA-Mn-zN-Z]' '[n-zN-Za-mA-M]'
    return 0
}
```

As you can see, **rot13** accepts command-line arguments which it passes via **echo** through a translate (**tr**) command that will print the result on **stdout**.

Entering **rot13 sheesh** produces **furrfu** on **stdout**, while the reversed **rot13 fuurfu** displays **sheesh**.

When to write a function

There are a lot of scripting situations where writing a short function of your own is a good idea.

Some of these include:

- Some common activity that will be used frequently
- Part of a larger script that will be repeated at least 2 or 3 times, perhaps slightly differently each time
- An uncommon activity used only once in a while, but you don't want to have to remember the details
- A tricky bit of logic – write once, use over and over, even if its not often
- A part of a large script that will only be used once
- The "Lego block" approach to scripting – develop functions that can be "plugged together" to form a complete script with a little "glue"

Sample function

```
START perms
    SOURCE file of functions
    SOURCE file of configuration info

    SET dir to arg 1 if present
    SET dir to . if no dir

    PUT config file values

    FOR each file in the directory
        CALL my-format with the current file name
    END FOR

    EXIT 0
END perms
```

Sample function

```
START my-format USING $1
     COMMAND for directory/file
     EXTRACT permissions
     PUT directory/file and permissions
     RETURN 0
END my-format
```

Data Dictionary (Global)

Name	Type	Value	Purpose
CMD	string	**ls -ld**	Command to execute
DELIM	string	**" "**	Delimiter for **cut**
DIR	string	undefined	Directory to use
FIELD	integer	**1**	Field number for **cut**

Sample function

In file **./config**:

```
CMD='ls -ld'
FIELD=1
DELIM=' '
```

In file **./include:**

```
function my-format ()
{
    echo file \"$DIR/$1\" permissions \
        $($CMD $DIR/$1 \
            | cut -f $FIELD -d "$DELIM")
    return 0
}
```

Note file permissions:

```
-rw-rw-r--. 1 allisor allisor  31 date config
-rw-rw-r--. 1 allisor allisor 121 date include
```

```
#! /bin/bash

# source my files
source ./include   # function here
source ./config    # variables here

# set default directory to $1 or .
DIR=${1:-$DIR}
DIR=${DIR:-.}

# show config values
echo Config: CMD = \"$CMD\"\; \
     FIELD = \"$FIELD\"\; DELIM = \"$DELIM\"

# do some stuff with the function
for x in $(ls $DIR); do
    my-format $x
done

exit 0
```

Execution of the script

```
System Prompt$ ./perms
Config: CMD = "ls -l"; FIELD = "1"; DELIM = " "
file "./config" permissions -rw-rw-r--.
file "./include" permissions -rw-rw-r--.
file "./perms" permissions -rwxrwxr-x.
System Prompt$
```

What's happening?

Both **./include** and **./config** are brought into the **perms** script as though they had been written there.

Then each time through the **for** loop, the next filename is passed to the function **my-format**. Once it has done its work, control returns to the calling script, **perms**, where it goes through the loop again until the list is exhausted.

The `getops` Function

- Many scripts depend heavily upon command line arguments, or positional parameters, so writers frequently use the **`getopts`** function to make processing them easier.
- It's often used for the **`-x`** form (the "dash" form) of options because of all the many combinations.
- For example, consider a command which can have options **`-x`**, **`-y`**, and **`-z`** in any order and any combination (like **`-xz`** and **`-zx`**), giving at least 29 (I counted) valid combinations. You don't want to write script code for each possibility!

getops

`getopts optionstring variable [argumentlist]`

- It will use the positional parameters unless the optional argumentlist is supplied.
- **`getopts`** puts the matching option character into your variable and puts the number of the next char into **`OPTIND`**.
- If the option is not matched, variable is set to "**`?`**" and an error message is written to **`stderr`** (unless "**`:`**" is the first character of optionstring to suppress that behaviour).
- Dash arguments that have arguments themselves (like **`cut -f 5`**) can be followed by a colon in optionlist, and the next item in the parameter list will be assigned to **`OPTARG`** for use in the script.
- Processing stops when the first non-dash argument is found.

getops Example (./goe)

```
#! /bin/bash
    declare  opt_char
    while getopts :xy:z opt_char; do
```

```
while getopts :xy:z opt_char; do #repeated
    case $opt_char in
        x) echo $opt_char found
           ;;
        y) echo $opt_char found \
           with \"$OPTARG\"
           ;;
        z) echo $opt_char found
           ;;
        *) echo option error index \
           $(( $OPTIND - 1 )) = \"$opt_char\"
           exit 1
           ;;
    esac
done
```

```
if (( $OPTIND > $# )); then
   echo All args have been processed
else
    echo First non-dash arg is \
    \$$OPTIND = \"$(eval echo -n \$$OPTIND)\"
fi
exit 0
```

```
System Prompt$ ./goe
All args have been processed
System Prompt$ ./goe x
First non-dash arg is $1 = "x"
System Prompt$ ./goe -x
x found
All args have been processed
System Prompt$ ./goe -x -y some-stuff -z
x found
y found with "some-stuff"
z found
All args have been processed
System Prompt$ ./goe -x more-stuff
x found
First non-dash arg is $2 = "more-stuff"
```