CS 114 - Fall 2004

Lecture 10 - Wednesday, October 20, 2004

Shell scripts

Shell functions

Commonly used commands can be put into functions. For example, the `make-thumbnail` script from last time printed an error message and usage information and then exited the script in two or three places. This can can be factored out and put into a function as follows:

```
usage() {
    echo "`basename $0`: $@"
    echo "usage: `basename $0` [-xy MxN] [-v] files..."
    exit 1
}
```

and invoked just like a normal command. For example:

```
usage "missing argument to -xy"
```

will print:

```
make-thumbnail: missing argument to -xy"
usage: make-thumbnail [-xy MxN] [-v] files...
```

and then exit the script with exit status 1, indicating failure.

Arguments passed to shell functions are in the variables `$1`, `$2`, ..., just as if the function were a separate shell script. However, the script runs *in the same shell*. So, invoking `exit` in a function will exit the script, and changing an environment variable will change that variable for the rest of the script.

The following script demonstrates the effect of a function on shell variables.

```
#!/bin/sh

f() {
    x=2
    y=3

    echo in the function $FUNCNAME
    echo '$0 =' $0
    echo '$1 =' $1
    echo '$@ =' $@
    echo '$* =' $*
    echo '$# =' $#
    echo '$x =' $x
    echo '$y =' $y
    echo returning
    true            # return 0; the exit status of the function
                    # is the exit status of the last command in
                    # the function
}
```

```
x=1

echo before the call
echo '$0 =' $0
echo '$1 =' $1
echo '$@ =' $@
echo '$* =' $*
echo '$# =' $#
echo '$x =' $x
echo '$y =' $y

# call rebinds $@, $#, $1, ..., but not $0
f p q

echo return status = $?

echo after the call
echo '$0 =' $0
echo '$1 =' $1
echo '$@ =' $@
echo '$* =' $*
echo '$# =' $#
echo '$x =' $x
echo '$y =' $y
```

Running the script produces the output:

```
before the call
$0 = scripts/test.sh
$1 = a
$@ = a b c
$* = a b c
$# = 3
$x = 1
$y =
in the function f
$0 = scripts/test.sh
$1 = p
$@ = p q
$* = p q
$# = 2
$x = 2
$y = 3
returning
return status = 0
after the call
$0 = scripts/test.sh
$1 = a
$@ = a b c
$* = a b c
$# = 3
$x = 2
$y = 3
```

Note that the parameters `$1`, ... are restored after the call returns, but any changes made to variables in the function are visible after the call returns.

Exit status

While a script can branch on the exit status of a command with `if` or `while`, the status can also be

checked directly. The variable `$?` contains the exit status of the previous command executed by the shell. Thus:

```
$ head file
head: file: No such file or directory
$ echo $?
1
$ echo hello > file
$ head file
hello
$ echo $?
0
```

For a pipeline, `$?` is set to the exit status of the last command in the pipeline.

```
$ head file2 | tail
head: file2: No such file or directory
$ echo $?
0
```

Even though the `head` command failed, the `tail` command succeeded and `$?` is set to 0.

Error messages

So far, the scripts we've been writing have printed error messages to `stdout`. But, most programs output error messages to another output stream `stderr`. By default, both `stdout` and `stderr` are sent to the console. Thus if I, for example, try to `cat` a non-existent file,

```
$ head file
head: file: No such file or directory
```

But what if I want to redirect the output of a command to a file, or to a pipe?

```
$ head file > file.out
head: file: No such file or directory
$ head file | grep hello
head: file: No such file or directory
```

The error messages are still sent to the console. The operator `>` will only redirect `stdout`. In the Bourne shell, error messages can be redirected to a separate file by using the operator `2>`. The number 2 is the *file descriptor* for `stderr`. Thus:

```
$ head file > file.out 2> file.err
$ cat file.err
head: file: No such file or directory
```

The first line redirects `stdout` to `file.out` and `stderr` to `file.err`. `cat file.err` sends the contents of `file.err` to `stdout` as usual.

If you want to suppress error messages entirely, you can redirect to the file `/dev/null`:

```
$ head file
head: file: No such file or directory
$ head file 2> /dev/null
$
```

`stderr` can also be sent to `stdout` using the (rather awkward) operator `2>&1`. Here, 2 is the file descriptor for `stderr` and 1 is the file descriptor for `stdout`. Redirecting `stderr` to `stdout` is useful

when we want to send error messages to the next command in a pipeline. For example:

```
$ head file1 file2
head: file1: No such file or directory
head: file2: No such file or directory
$ head file1 file2 2>&1 | grep 'file2'
head: file2: No such file or directory
```

In the second command, both error messages are redirected to `stdout`, then sent through the pipe to `grep`, which prints only the error for `file2`.

The redirection can go the other way as well. To redirect `stderr` to `stdout`, use the operator `1>&2`, or just `>&2`. Thus, if we want our script error messages sent to `stderr` instead of to `stdout`, the `usage` function above could be written:

```
usage() {
    echo "`basename $0`: $@" >&2
    echo "usage: `basename $0` [-xy MxN] [-v] files..." 1>&2
    exit 1
}
```

Error redirection in the C shell

In the C shell, redirection is a bit less flexible: there is now way to redirect `stderr` to a different file than `stdout`. The best that can be done is:

```
% head file >& file.out
```

This sends *both* `stdout` and `stderr` to the file `file.out`. In the following command, both streams are redirected through a pipe to `grep`:

```
% head file1 file2 |& grep 'file2'
head: file2: No such file or directory
```

Example: a simple file locator

Below is a pair of scripts that I use to more efficiently locate files in my home directory. The first script builds a database of all files in the user's home directory.

```
#!/bin/sh
# mkslocatedb - Build a locate database for
# the user's home directory.

# the database
slocatedb=$HOME/.slocatedb

# Set the umask so the database file will be inaccessible
# to everyone else.
umask 077

# Create the database in a temporary file so the old database
# can continue to be used while the new one is built.
# The database is compressed as it is built.
#
# Note: we send stderr to /dev/null to suppress errors
# that might occur when find hits a directory it can't
# read.
find $HOME -print 2> /dev/null | gzip -c > $slocatedb.new.gz
```

```
# Move the DB into place
mv $slocatedb.new.gz $slocatedb.gz
```

The second script uses the database to lookup files.

```
#!/bin/sh
# slocate - lookup a file in the slocate DB

slocatedb=$HOME/.slocatedb

if [ $# -ne 1 ]; then
    echo "usage: `basename $0` pattern" >&2
    exit 1
fi

# Do the lookup if the database is there.
# Use egrep's \< and \> anchors to match the beginning
# and end of a word.
if [ -f $slocatedb.gz ]; then
    gunzip -c $slocatedb.gz | egrep "\<$1\>" | sort -u
fi
```

To use the two scripts, run the `mkslocatedb` script periodically to build or update the database. You can set up your system to run the script automatically every hour. Read the man page for `cron` and for `crontab` if you want to figure out how.

To lookup a file, just run `slocate` *`regex`* For example:

```
$ slocate hw4.tex
/Users/nystrom/114/handouts/hw4.tex
$ vi `slocate hw4.tex`
```

The first command returns all files in my home directory containing the string "`hw4.tex`" (actually files matching the regex `hw4.tex`). The second command opens the matching file in an editor.

Debugging shell scripts

Here are a few suggestions to help you debug a shell script:

- Start small. Get the basic functionality of the script working and then add features like command-line argument processing and error checking.
- If you're not sure what a command does, just try it on the command line. If the command expects to read from `stdin`, use `echo` to give it some input or redirect a file to it with `<` *`file`*.
- To check if a variable `var` is set as you expect, particularly in the presence of substitutions, insert `echo $var` into the script after `var` is set.
- To see what your script is doing as it runs, add `set -x` near the top of the script. This will cause `sh` to print out each command as it is run—with substitutions performed. You can turn this option off by adding `set +x` later.
- Use `#` to comment out a statement.