===================================
Reading stdin into shell variables
===================================

-IAN! idallen@idallen.ca

Many shell scripts need to obtain information from the keyboard or
standard input. You might need to prompt the user to enter a file name,
or to answer yes/no to a question. The built-in command that reads
stdin in the Bourne shells is named "read".

The "read" command reads one line of standard input, splits it on blanks
(and only on blanks - quotes and other meta-characters are not special)
and stores it into one or more shell variables named on the same
command line:

```
    $ read a b c
    one two three   four   five             # this is typed by the user
    $ echo "$a"
    one
    $ echo "$b"
    two
    $ echo "$c"
    three   four   five
```

The first blank-separated word goes into the first named variable, the
second into the second named variable, etc. for the first N-1 variable
names. All the words that are left over go into the Nth and last
variable, including any embedded blanks between those left over words.

```
    $ read a b c
    "one two" * four                        # this is typed by the user
    $ echo "$a"
    "one
    $ echo "$b"
    two"
    $ echo "$c"
    * four
```

The words going into the first N-1 named variables have no blanks in or
around them at all. Quotes and GLOB meta-characters have no meaning;
they are just ordinary characters. Blanks can only appear between the
left-over words put into the last named variable. If you give only one
named variable to "read", the entire line of standard input is copied
into the one variable unchanged, including all the embedded blanks:

```
    $ read x
    "one two"   *   four                    # this is typed by the user
    $ echo "$x"
    "one two"   *   four
```

Note that the arguments to the "read" command are variable names, e.g.
"read foo", not expansions of variables, e.g. NOT "read $foo". We don't
want the shell to expand the variable "$foo" before running the "read"
command; we want the "read" command to get the name of the variable,
"foo", directly. Don't use dollar signs on the "read" command line
variable names!

If there are not enough words to fill all the named variables on the
"read" command line, the extra variables are set to be empty (null

string):

```
    $ read a b c
    one                                        # this is typed by the user
    $ echo "/$a/$b/$c/"
    /one///                                    # $b and $c are empty strings
```

Entering a blank line in response to a "read" will set *all* the named
variables to be empty:

```
    $ read a b c
                                               # user enters a blank line
    $ echo "/$a/$b/$c/"
    ////
```

Return status and EOF

The "read" command, like all shell commands, has a return status.
It returns 0 (good) if the read succeeds. (The read succeeds even if you
enter a blank line!) It returns 1 (failed) only on end-of-file (EOF).
The EOF signal means "read" can be used to read many lines of standard
input from a file:

```
    #!/bin/sh -u
    n=0
    while read line ; do
       let n=n+1
       echo "Line $n contains $line"
    done
```

The above loop can be placed into a shell script whose input is coming
from a file, e.g. "./script </etc/passwd", and the script will loop
reading lines from the file until EOF. At EOF, the "read" command will
return a bad status and the WHILE loop will end.

An EOF response to a "read" causes all the named variables on the "read"
command line to be set to the null string, just as if you had entered
a blank line. Only EOF causes "read" to return a non-zero status.

As with all Unix commands, you can signal EOF from the keyboard by
typing the EOF character at the beginning of a line (usually CTRL-D).

(mis-)Using "read" in a pipe

The shells start up separate processes to handle the various parts of
a shell pipeline, e.g.

```
    $ echo one two three | read a b c
```

The "echo" command and the "read" command are connected by a pipe and
are running in different processes than the current shell. The "read"
command is reading standard input into variables *in a different process*.
This means that the variables a,b,c in the current interactive shell
are unaffected by the "read":

```
    $ a=ant b=bat c=cow
    $ echo "/$a/$b/$c/"
    /ant/bat/cow/
    $ echo one two three | read a b c
    $ echo "/$a/$b/$c/"
    /ant/bat/cow/
```

You cannot pipe input into "read" and expect it to affect variables in the current shell. Of course, file redirection doesn't suffer from the same multi-process problem (no separate process is created to handle file redirection):

```
    $ a=ant b=bat c=cow
    $ echo one two three >tmp
    $ read a b c <tmp
    $ echo "/$a/$b/$c/"
    /one/two/three/
```

Because no second process is created to handle file redirection, the "read" executes in the current shell. It reads one line of input from standard input, which in the above example comes from the file "tmp". The one line of input is split on blanks and is assigned to the variables a,b,c in the current shell.

Prompt before you read

Shell scripts that use "read" to read from standard input should issue a prompt to indicate that some input is expected from the keyboard:

```
    #!/bin/sh -u
    echo 1>&2 "$0: Enter number of seconds to sleep:"
    read seconds
    echo "You entered '$seconds' seconds; sleeping ..."
    sleep $seconds
```

Without the prompt to tell the user what input is needed, the user of the script has no idea what is expected.

As with error messages, prompts must be sent to standard error, not to standard output. Prompts sent to stdout disappear when output is redirected, and the user again has no idea what is wrong! Prompts on stderr are visible even with redirection:

```
    $ ./script >out
    ./script: Enter number of seconds to sleep:
    2
    $
    $ cat out
    You entered '2' seconds; sleeping ...
```

Prompting only if reading from a keyboard [OPTIONAL]

Prompts are only useful if standard input is coming from a keyboard, where the user is expected to type something. If a script is reading standard input from a pipe or a file, no prompt is needed. In fact, the prompt is incorrect and misleading:

```
    $ echo 2 | ./script
    ./script: Enter number of seconds to sleep:
    You entered '2' seconds; sleeping ...
```

The prompt appears even though standard input will come from the pipe
and the user gets no chance to enter anything from the keyboard.

The "tty" command is useful to determine if standard input is coming from
a terminal. It prints the terminal device pathname and returns a command
status of true (zero) if standard input is connected to a terminal device
(i.e. to your keyboard):

```
    $ tty                         # tty command reads from keyboard
    /dev/pts/1
    $ echo $?
    0

    $ echo hi | tty               # tty command reads from pipe, not keyboard
    not a tty
    $ echo $?
    1

    $ tty </etc/passwd            # tty command reads from file, not keyboard
    not a tty
    $ echo $?
    1

    $ tty </dev/null              # tty command reads /dev/null, not keyboard
    not a tty
    $ echo $?
    1
```

The "-s" option to tty suppresses the output (you could also redirect
output to /dev/null), allowing us to use only the return status in
a script. The following modified script prompts only if standard input
is connected to a keyboard:

```
    #!/bin/sh -u
    # only prompt if stdin of this script is coming from a keyboard
    if tty -s ; then
        echo 1>&2 "$0: Enter number of seconds to sleep:"
    fi
    read seconds
    echo "You entered '$seconds' seconds; sleeping ..."
    sleep $seconds
```

Now, the prompt appears only if the user needs to type something on
the keyboard; when input is redirected (no keyboard), no prompt appears:

```
    $ ./script
    ./script: Enter number of seconds to sleep:
    2
    You entered '2' seconds; sleeping ...

    $ echo 3 | ./script
    You entered '3' seconds; sleeping ...

    $ echo 4 >file
    $ ./script <file
```

```
You entered '4' seconds; sleeping ...
```