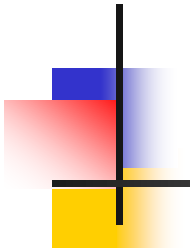

System Programming

Advanced Shell Scripting

Creating and Using Functions

- The formal definition of a shell function is as follows:

  name () { *list* ; }

- Valid and invalid function definitions:

  lsl() { ls -l ; } # valid

  lsl { ls -l ; } # invalid

- Aliases definition for sh:

  $> cat mycd

  cd () { chdir ${1:-$HOME} ; PS1="`pwd`$ " ; export PS1 ; }

- $> source mycd

Function Examples

- Listing the current value of PATH, with each directory listed on a single line.

```
lspath() {
        OLDIFS="$IFS"
        IFS=:
        for DIR in $PATH ; do echo $DIR ; done
        IFS="$OLDIFS"
}
```

- $> lspath | grep "/usr/dt/bin"

Function Examples (Cont.)

- Tailoring Your Path

```
setPath() {
        PATH=${PATH:="/sbin:/bin"};
        for _DIR in "$@"
        do
          if [ -d "$_DIR" ] ; then
                        PATH="$PATH":"$_DIR" ; fi
        done
        export PATH
        unset _DIR
}
```

Function Examples (Cont.)

- An example invocation:

  $> setPath /sbin /usr/sbin /bin /usr/bin /usr/ccs/bin

- It checks to see whether each of its arguments is a directory, and if a directory exists, it is added to PATH.

Function parameter passing

- A parameter is passed to a function as it is passed to shell script.
- The syntax to define function:

```
function function-name( )
{
    statement1
    statement2
    statementN
}
```

- This function is called from command line or within the shell script as follows:

```
function-name arg1 arg2 arg3 argN
```

Parameter passing example

- $ vi pass

```
function demo()
{
  echo "All Arguments to function demo(): $*"
  echo "First argument $1"
  echo "Second argument $2"
  echo "Third argument $3"
  return
}
# Call the function
demo -f foo bar
```

Return Value

- Example definition

```
function add_two {
    (( sum=$1+$2 ))
    return $sum
}
```

- Invoking the function

```
add_two 1 3
echo $?
```

- $? value returned by last function call or command
- Function definition must occur before the function is called in the script

Sharing Data Between Functions

- The C shell, csh, provides three commands for quickly moving around in the UNIX file system:

  popd pushd dirs

- These commands maintain a stack of directories internally and enable the user to add and remove directories from the stack and list the contents of the stack.

Implementing dirs

```
dirs() {
    # save IFS, then set it to : to access the
    # the items in _DIR_STACK individually.
    OLDIFS="$IFS"
    IFS=:
    # print each directory followed by a space
    for i in $_DIR_STACK
    do
        echo "$i \c“
    done
    ...
```

Implementing dirs (Cont.)

```
...
# add a new line after all entries in
# _DIR_STACK have been printed
echo
# restore IFS
IFS="$OLDIFS"
}
```

Implementing pushd

```
pushd() {
    # set the requested directory, $REQ, to the first
    argument
    # If no argument is given, set REQ to .
    REQ="$1";
    if [ -z "$REQ" ] ; then REQ=. ; fi
    # if $REQ is a directory, cd to the directory
    # if the cd is successful update $_DIR_STACK
    # otherwise issue the appropriate error messages
    if [ -d "$REQ" ] ; then
            cd "$REQ" > /dev/null 2>&1
    ...
```

Implementing pushd (Cont.)

```
    ...
        if [ $? -eq 0 ] ; then
                _DIR_STACK="`pwd`:$_DIR_STACK" ;
                export _DIR_STACK ; dirs
        else
                echo "ERROR: Cannot change to
    directory                       $REQ." >&2
        fi
    else
    echo "ERROR: $REQ is not a directory." >&2
    fi
    unset REQ
}
```

Implementing popd

```
_popd_helper() {
    # set the directory to pop to the first argument, if
    # this directory is empty, issue an error and return 1
    # otherwise get rid of POPD from the arguments
    POPD="$1"
    if [ -z "$POPD" ] ; then
        echo "ERROR: The directory stack is empty."
    >&2
        return 1
    fi
    shift
    ...
```

Implementing popd (Cont.)

```
...
# if any more arguments remain, reinitalize the directory
# stack, and then update it with the remaining items,
# otherwise set the directory stack to null
if [ -n "$1" ] ; then
        _DIR_STACK="$1" ;
        shift ;
        for i in $@ ; do
_DIR_STACK="$_DIR_STACK:$i"      ; done
else
        _DIR_STACK=
fi
...
```

Implementing popd (Cont.)

```
...
# if POPD is a directory cd to it, otherwise issue
# an error message
if [ -d "$POPD" ] ; then
        cd "$POPD" > /dev/null 2>&1
        if [ $? -ne 0 ] ; then
                echo "ERROR: Could not cd to $POPD."
>&2
        fi
        pwd
else
        echo "ERROR: $POPD is not a directory." >&2
fi
...
```

Implementing popd (Cont.)

```
    ...
    export _DIR_STACK
    unset POPD
}
popd() {
    OLDIFS="$IFS"
    IFS=:
    _popd_helper $_DIR_STACK
    IFS="$OLDIFS"
}
```

echo command

- echo display text or value of variable.

  echo [options] [string, variables...]

- Options

  -n Do not output the trailing new line.

  -e Interpret the following escaped chars.

  \c suppress trailing new line

  \a alert (bell) \b backspace

  \n new line \r carriage return

  \t horizontal tab \\ backslash

- **$** echo -e "An apple a day keeps away \a\t\tdoctor\n"

Displaying colorful text

- There are some control chars used with echo
- This commad prints message in Blue color.

  $> echo "\033[34m Hello Colorful World!"
  Hello Colorful World!
- This uses ANSI escape sequence (\033[34m).
  - **\033**, is escape character, takes some action
  - **[34m** escape code sets foreground color to Blue
    - **[** is start of CSI (Command Sequence Introduction).
    - **34** is parameter.
    - **m** is letter (specifies action).
- General syntax

  echo -e "**\033[***escape-code your-message*"

Displaying colorful text

- A list of *escape-code/action letter or char.*

Char.	**Use in CSI**
h	Set the ANSI mode
l	Clears the ANSI mode
m	Show characters in different **colors** or effects such as **BOLD** and Blink
q	Turns keyboard num lock, caps lock, scroll lock LED on or off
s	Stores the current cursor x, y position (col , row position) and attributes
u	Restores cursor position and attributes

Displaying colorful text

- m understands following parameters.

Param.	Meaning
0	Sets default color scheme (White foreground and Black background), normal intensity, no blinking etc.
1	Set **BOLD** intensity

```
$> echo –e "I am \033[1m BOLD \033[0m Person“
```

I am **BOLD** Person

2	Set dim intensity

```
$> echo –e "\033[1m  BOLD \033[2m DIM  \033[0m“
```

BOLD DIM

Displaying colorful text

5 Blink Effect

```
$> echo –e "\033[5m Flash!  \033[0m“
```

Flash!

7 Reverse video effect i.e. Black foreground and white background by default

```
$> echo -e "\033[7m Linux OS! Best OS!! \033[0m“
```

Linux OS! Best OSl

Displaying colorful text

25 Disables blink effect

27 Disables reverse effect

30 – 37 Set foreground color

31->Red, 32->Green, ...

```
$> echo -e "\033[31m I am in Red“
```

I am in Red

40 – 47 Set background color

```
$> echo -e "\033[44m Wow!!!"
```

Wow!!!

Displaying colorful text

- q understand following parameters

Param.	Meaning
0	Turns off all LEDs on Keyboard
1	Scroll lock LED on and others off
2	Num lock LED on and others off
3	Caps lock LED on and others off

Script execution

- Provide script as an argument to the shell program (e.g. `bash my_script`)
- Or specify which shell to use within the script
  - First line of script is `#!/bin/bash`
  - Make the script executable using `chmod`
  - Make sure the PATH includes the current directory
  - Run directly from the command line
- No compilation is necessary!