*AIX Ver. 4*
*Korn Shell Programming*
(Course Code AU23)

Master Visuals
ERC2.2

IBM Learning Services
Worldwide Certified Material

**Publishing Information**

This publication has been produced using BookMaster® (Program Number 5668-015), the Document Composition Facility (Program Number 5748-XX9), and Freelance for Windows. It was printed on the IBM® PostScript Printer.

## Trademarks

IBM® is a registered trademark of International Business Machines Corporation.

The following are trademarks International Business Machines Corporation in the United States, or other countries, or both:

| | | |
|---|---|---|
| AIX | AT | BookMaster |
| IBM | OS/2 | RISC System/6000 |
| RS/6000 | | |

Freelance and Notes are trademarks of Lotus Development Corporation in the United States, or other countries, or both.

Windows is a registered trademark of Microsoft Corporation.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Other company, product, and service names may be trademarks or service marks of others.

**July 2000 Edition**

The information contained in this document has not been submitted to any formal IBM test and is distributed on an “as is” basis without any warranty either express or implied. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will result elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

# Contents



## Units



# Course Presentation Material Overview

Included in this package are landscape, black and white paper copies of each of the student visuals included in the Student Notebook. The paper copies are to be used to reproduce overhead transparencies required to teach the course. It is recommended that black-on-clear transparencies be used to reproduce the package.

In addition to the student visuals, a welcome visual SHOULD be included. To ensure that each student is attending the correct course for which they originally enrolled, the welcome visual should be used to provide the student with the course name and course code upon entering the classroom.

It is advised that upon checking out the classroom prior to the start of class, there be two overhead projectors in the classroom. This will guarantee that there is a working overhead projector available at all times for the duration of the class.

The paper copies and the transparencies created from the paper copies are the property of IBM. By way of protecting our intellectual properties, neither the paper copies nor the transparencies should be given to anyone other than a course certified instructor. The copies are not to be used for any purpose other than teaching the course. The security of this package and the products created as a result of this package are the responsibility of the course certified instructor.

# Unit 1. Basic Shell Concepts

# Objectives

To review basic Shell concepts in order to:

- Describe the AIX Shells
- Use the AIX file-system
- Create a Shell Script
- Use metacharacters
- Use I/O redirection
- Use pipes and tees
- Group commands
- Run background processes
- Use Korn Shell job control
- Use command line recall and editing

# Shells

## What is a Shell ?

- User interface to AIX
- Command interpreter
- Programming language

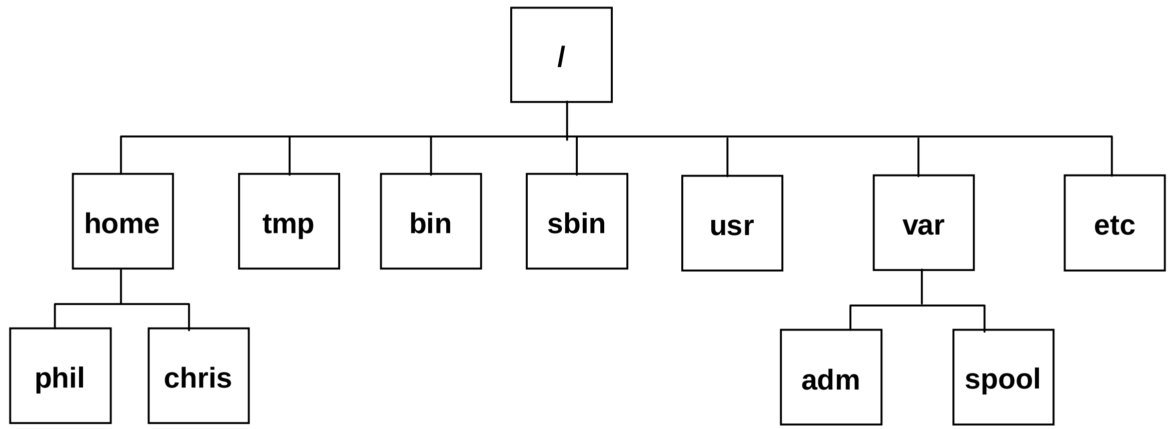
### AIX Shells:

| | | |
|---|---|---|
| • Korn | - ksh | |
| • Bourne | - bsh | |
| • Restricted | - Rsh | |
| • C | - csh | |
| • Trusted | - tsh | |
| • POSIX | - psh | |
| • Default | - sh | *link to ksh in AIX V4* |
| • Remote | - rsh | |

AU232102

# Directories

The file-system comprises directories in a hierarchical structure




AU232104

# A File

**Definition:**

- collection of data, located on a portion of a disk.
- stream of characters or a "byte stream".

No structure is imposed on an ordinary file by the operating system.

**Examples:**

- Binary executable code – /bin/ksh
- Text data – /etc/passwd
- C program text – /home/john/prog.c
- Device special file – /dev/null
- Directory special file – /home

**`$ file filename`** ***– to find out which file type***

# AIX File Names

- Should be descriptive of the content
- Are case-sensitive
- Should use only alphanumeric characters:

  UPPERCASE lowercase digits

  # . @ - _

- Should not begin with "+" or "-" sign
- Should not contain embedded blanks or tabs
- Should not contain shell "special" characters:

```
* ? > < / ; & ! ~
[ | ] $ \ ' " ` { } ( )
```

# What is a Shell Script?

- A readable text file which can be edited with a text editor
  - /usr/bin/vi *shell_prog*
- Anything that you can do from the Shell prompt
- A program, containing:
  - System commands
  - Variable assignments
  - Flow control syntax
  - Shell commands

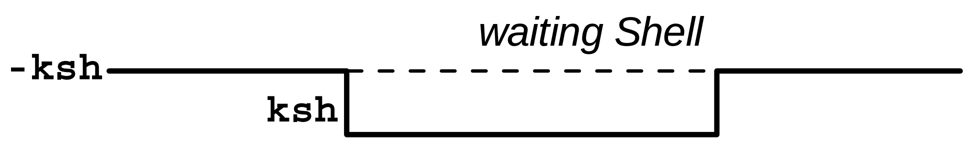
**and Comments !**

# Invoking Shells

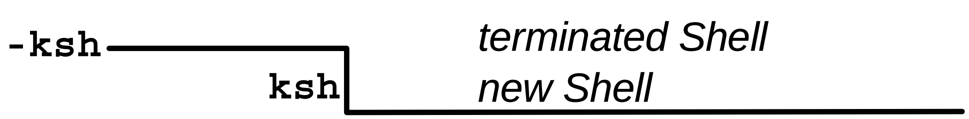
`$ ksh` *begins a new Shell, interrupting the current one*

`$ ksh -c commands` *runs commands in a Shell*

`$ ksh -r` *starts a restricted Shell*




`$ exec ksh` *terminates the current Shell and replaces with new Shell*

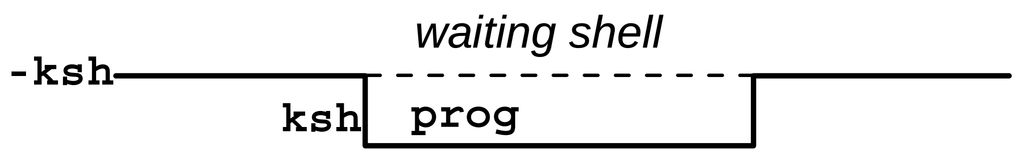


# Invoking Scripts

`$ . prog` — *prog run (sourced) in **current** Shell environment*


`$ ksh prog` — *run prog in a new Shell*

`$ prog` — *run in a new Shell if prog is executable*




`$ exec prog` — *run in a new Shell to replace the current one*

# Korn Shell Configuration Files

Invoking the Korn Shell sources:

| | |
|---|---|
| /etc/environment | Sourced by all AIX processes |
| /etc/profile | Sourced by login Shells |
| .profile | Login Shells source this file in the user's home directory |
| ENV file | A resource file listed in the ENV Environment Variable will be sourced by Korn Shells |

time

Each new **explicit** Korn Shell sources the ENV file again

AU232116

# What Are Metacharacters?

Characters with special meaning

- 3 types
  - Wildcard (or expansion)
  - Korn Shell
  - Quoting

- Shell processes metacharacters before executing a command

- There are several different Shell metacharacters

- Metacharacters can be mixed

They can be turned off by Shell options

# Wildcard Metacharacters

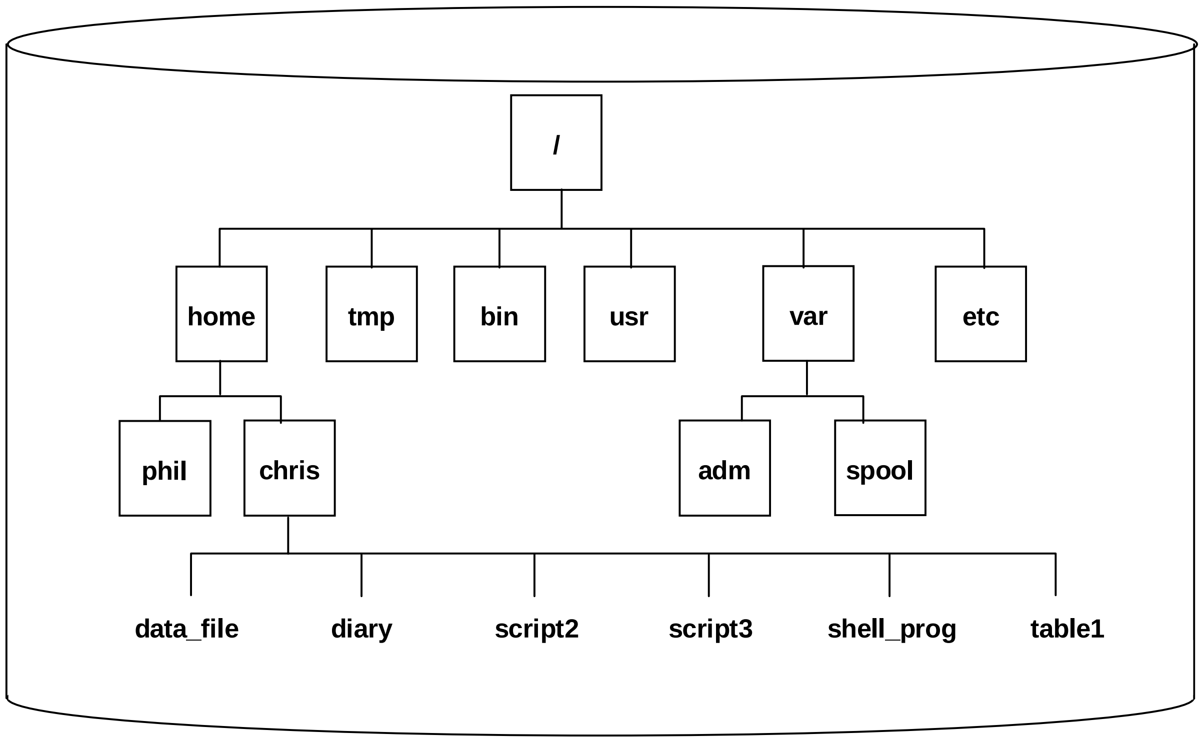
Metacharacters that form patterns that are expanded into matching filenames from the current directory

| | | |
|---|---|---|
| ***** | - | Match any number of any characters |
| **?** | - | Match any single character |
| **[abc]** | - | Match a single character from the bracketed list |
| **[!az]** | - | Match any single character except those listed |
| **[a-z]** | - | Inclusive range for a list |

**Character Equivalence Classes** can be used in place of range lists, to avoid National Language collation problems:

| | | |
|---|---|---|
| [**[:upper:]**] | - | range list of all upper case letters |
| [**[:lower:]**] | - | all lower case letters: a, b, c,... z |
| [**[:digit:]**] | - | digits: 0, 1, 2,... 9 |
| [**[:space:]**] | - | spacing characters: tab, space, etc. |

# Sample Directory







AU232122

# Expansion Examples

| | |
|---|---|
| `$ rm d*y` | *removes the diary file* |
| `$ file script*` | *identifies script2 and script3* |
| `$ head script[345]` | *displays the top lines of script3* |
| `$ more script[3-6]` | *displays script3 screen by screen* |
| `$ tail script[!12]` | *displays the last lines of script3* |

Now your turn...

```
$ touch ?a*

$ pg [st][ah]*

$ lpr [a-z]*t[0-9]
```

# Korn Shell Metacharacters

## The Korn Shell can match multiple patterns

| | |
|---|---|
| `*(pattern|pattern...)` | zero or more occurrences |
| `?(pattern|pattern...)` | zero or one occurrence |
| `+(pattern|pattern...)` | one or more occurrences |
| `@(pattern|pattern...)` | exactly one occurrence |
| `!(pattern&pattern...)` | anything except |

One or more patterns, separated with "|" for "or", "&" for "and"

## Examples:

| | |
|---|---|
| `*([0-9])` | *0 or more consecutive digits* |
| `?(warning)` | *0 or 1 occurence of "warning"* |
| `+([[:upper:]]|[a-z])` | *1 or more consecutive letters* |
| `@([0-9]|abc)` | *1 digit or "abc"* |
| `!(err*&fail*)` | *Word cannot start with "err" or "fail"* |

# Quoting Metacharacters

Stops normal Shell metacharacter processing, including metacharacter expansion

- To form strings

| | |
|---|---|
| **"double quotes"** | group characters into a string, and allow variable and command substitution |

- To form literal strings

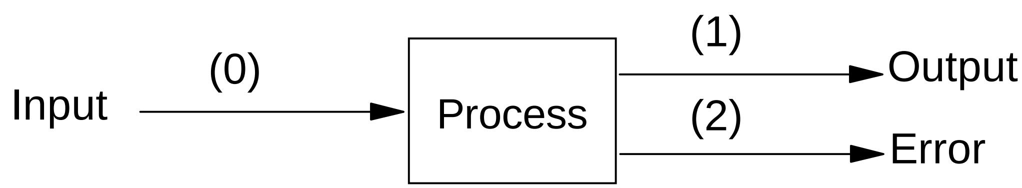
| | |
|---|---|
| **'single quotes'** | remove any special meaning for the characters within them |

- For a literal character

| | |
|---|---|
| **\character** | removes the special meaning of the character following the \ |

# Process I/O

- Every process has a file descriptor table associated with it




File descriptor table

| | | | |
|---|---|---|---|
| Defaults | 0< | Standard in | - keyboard |
| | 1> | Standard out | - screen |
| | 2> | Standard error | - screen |
| User-defined | 3 | | |
| | ⋮ | | |
| | 9 | | |




AU232130

# Input Redirection

Redirecting standard input from a file: <

command < filename

```
$ mail gene
Subject: Hello
A letter to see if you are still with us.
<Ctrl-d>
$ _

$ mail -s "Hello" gene < letter
$ _
```

Input may also be given inline. This is called a HERE document.

command << END
text
…
END




AU232132

# Output Redirection

Redirecting standard output to a file: >

command > filename

```
$ ls /home/chris
data_file script2 script3 shell_prog table1
$ _

$ ls /home/chris > listing
$ _
```

Redirecting standard error output to a file: 2>

command 2> filename

```
$ cat /home/chris/printout
cat: 0652-050 Cannot open printout.
$ _

$ cat /home/chris/printout 2> errors
$ _
```

# Output Appending

Appending standard output to a file: >>

command >> filename

```
$ wc -l /home/chris/script3
     42  /home/chris/script3
$ _

$ wc -l /home/chris/script3 >> line_count
$ _
```

Appending standard error output to a file: 2>>

command 2>> filename

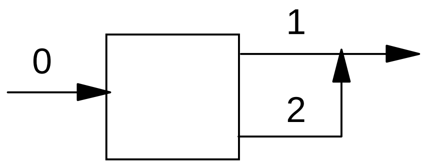
```
$ wc -c /home/chris/characters
wc: 0652-755 Cannot open characters.
$ _

$ wc -w /home/chris/words/ 2>> errors
$ _
```

# Association

File descriptors can be joined, so that they output to the same place

command > file 2>&1




Redirects standard error to join with standard out

What do you think this command does?

```
$ cat Message_file 1>&2
```

# Setting I/O or File Descriptors

The built-in Shell command exec allows you to

- open
- associate
- close

file descriptors

| | |
|---|---|
| `$ exec n> of` | *Opens output file descriptor n to file "of"* |
| `$ exec n< if` | *Opens input file descriptor n to read file "if"* |
| `$ exec m>&n` | *Associates output file descriptor m with n* |
| `$ exec m<&n` | *Associates input file descriptor m with n* |
| `$ exec n>&-` | *Closes output file descriptor n* |
| `$ exec n<&-` | *Closes input file descriptor n* |

# Setting I/O Descriptor Examples

To open file descriptor 3 for output to Dale's out file and 4 to Dale's err file

```
$ exec 3> /home/dale/out
$ exec 4> /home/dale/err
$ date >&3
$ ls /home/gale >&4
```

To associate output to file descriptor 3 with file descriptor 4

```
$ exec 3>&4
$ wc -l /home/gale/script3 >&3
$ wc -l /home/gale/table1 >&4
```

To close file descriptors 3 and 4

```
$ exec 3>&-
$ exec 4>&-
```

# Pipes

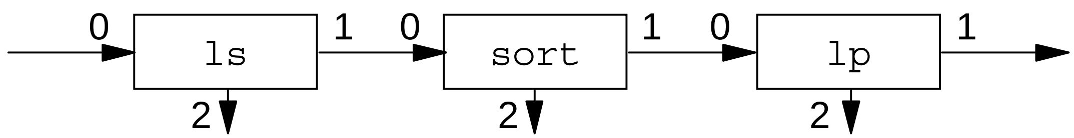
Commands can be joined, so one inputs into the next

```
command1 | command2 | command3
```

Gives a command *pipeline*

```
$ ls /home/robin | sort -r  |  lp
```

*sorts the file list into reverse order, and prints it*




Pipelines may have a branch using *the tee* command

- duplicates the standard input to the branch and to standard out

```
$ ls /home/francis | tee raw_list | sort -r | lp
```

*saves the unsorted list in the file raw_list*

# Command Grouping ( )

To combine the output of several commands: ( ) or { }

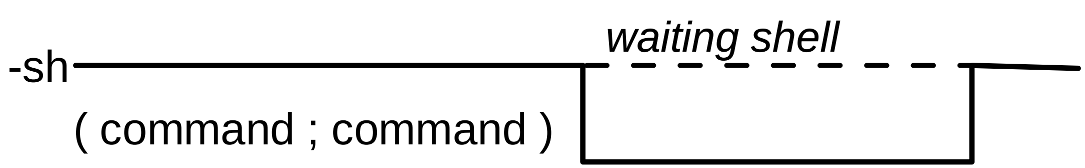
**( command ; command ... )**

- Runs commands in a Sub-Shell

For root to alter Lynn's files:

```
# ( cd /home/lynn ; chown lynn:bin d* )
```

leaves the working directory unchanged on completion



# Command Grouping {}

**{ command ; command ... ; }**

- Runs commands in the current Shell
- Directory (or environment) changes remain in effect
- Must leave spaces around the braces

| | |
|---|---|
| Either | have the braces on separate lines |
| or | include a final "; " before the closing brace |

```
# { cd /home/lynn ; chown lynn:bin s* ;}
```

{ command ; command ; }

-ksh

# Background Processing

Execute command in the background: &

command &

```
$ sleep 999  &
```

Waiting for the end...

```
$ date
Fri Dec 31 11:59:59 EST 1999
$ wait
```

*When all background processes have finished*

```
$ _
```

# Korn Shell Job Control

Korn Shell assigns job numbers to background or suspended processes

- The **jobs** command lists your current Shell processes and their job ids
- **Ctrl-z** suspends the current foreground job
- **bg** runs a suspended job in background
- **fg** brings to foreground a suspended or background job
- Jobs can be stopped with the **kill** command

kill, fg and bg work with the following arguments:

| | |
|---|---|
| `pid` | process id |
| `%job_id` | job id |
| `%%` - or - `%+` | current job |
| `%-` | previous job |
| `%command` | match a command name |
| `%?string` | match string in command line |

# Job Control Example

```
$ cc -o RUNME program_in.c
...
```

*After some time running this long compilation...*

```
Ctrl-z
[2] + 5692 Stopped (SIGTSTP)  cc -o RUNME
program_in.c
$ jobs
+ [2] Stopped (SIGTSTP)       cc -o RUNME
program_in.c
- [1] Running                 sleep 999 &
$ bg %+
[2] cc -o RUNME program_in.c
$ jobs
+ [2] Running                 cc -o RUNME
program_in.c
- [1] Running                 sleep 999 &
$ kill %cc
[2] + 5692 Terminated         cc -o RUNME
program_in.c
$ fg %1
sleep 999
$ _
```

*Completing the sleep in the foreground...*

AU232164

# Command Line Editing and Recall

Vi option for the Korn Shell gives:

- Command line editing
- Command recall

```
$ set -o vi
```

Then simply press **ESC** to enter editing mode:

- **h** to move the cursor left
- **l** to move the cursor right
- **-** or **k** fetches commands from the history file
- **+** or **j** if you go too far back
- Plus other *vi* commands to perform line editing

# Summary

- AIX Shells
- Hierarchical file-system
- File names and types
- Shell Scripts
- Invoking Shells
- Shell metacharacters: expansion, Korn and quoting
- < and << input redirection
- > and >> output redirection
- 2> and 2>> error redirection
- Setting file descriptors
- Pipes and tees
- Command grouping
- Background processes
- Korn Shell job control
- Korn Shell command editing

# Unit 2. Variables

# Objectives

How to use Shell variables and parameters:

- Setting variables
- Referencing variables
- Using Positional Parameters
- Shifting arguments
- Setting Positional Parameters
- Using Shell parameters
- How inheritance works
- Listing Shell variables
- Listing Environment variables



# Setting Variables

To assign a value to a variable: **name=value**

```
$ var1=Fri
$ _
```

To protect a variable against further changes:

**`readonly name=value`**

**- or -**

**`typeset -r name=value`**

```
$ readonly var1=Sun
$ var1=Mon
ksh: var1: This variable is read only
$ _
```



# Referencing Variables

To reference a variable, prefix name with a **$**

```
 $ print $var1
Fri
$ _
```

To separate a variable reference from other text use: **${ }**

```
 $ print The course ends on $var1day
The course ends on
$ print The course ends on ${var1}day
The course ends on Friday
$ _
```

# Positional Parameters

Parameters can be passed to Shell Scripts as arguments on the command line

```
$ params.ksh arg1 arg2
```

- "arg1" is Positional Parameter number 1
- "arg2" is Positional Parameter number 2
- Others are unset

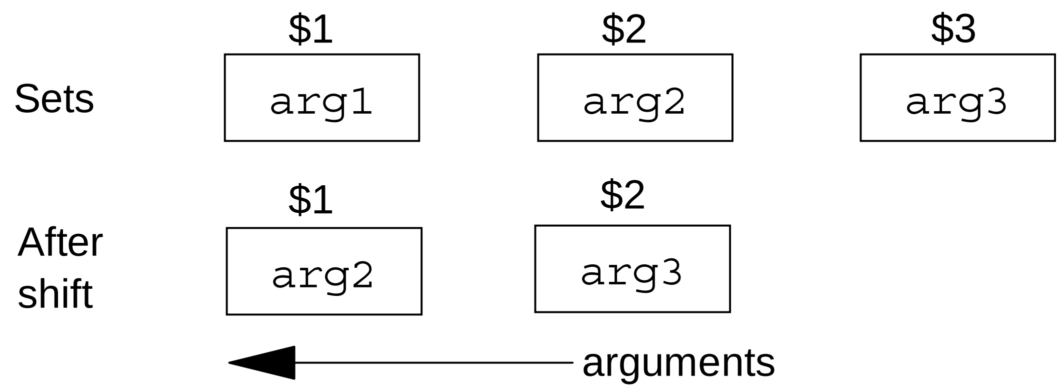
They are referenced in the script by:

- $1 to $9 for the first nine
- ${10} to ${n} for the remainder (Korn Shell only!)

# Shifting Arguments

In a Shell Script the **shift** command moves arguments "to the left":

```
$ params.ksh arg1 arg2 arg3
```




- Discarding the first or "leftmost" argument
- Decrementing the number of Positional Parameters
- Allowing Bourne Shell to reference more than 9 arguments

# Setting Positional Parameters

In a Shell Script the **set** command can:

- Change the values of Positional Parameters
- Unset Positional Parameters previously set

```
 $ cat first.ksh
print $1 $2 $3
set value1 value2
print $1 $2 $3

$ first.ksh a b c
a b c
value1 value2

$ _
```

# Variable Parameters

Shell Scripts set a number of other Shell Parameters:

| | |
|---|---|
| $# | The number of Positional Parameters set |
| $@ | Positional Parameters in a space separated list |
| $* | Positional Parameters in a list separated by the first Field Separator (the default is a space) |

In double quotes, $@ and $* behave differently:

"$@" = `"$1" "$2" "$3" . . .`

"$*" = `"$1 $2 $3 . . . "`




AU232212

# Some Shell Parameters

Shell Parameters that remain fixed for the duration of the Script:

$0 The (path)name used to invoke the Shell Script

$$ The Process Id (PID) of current process (shell)

$- Shell Options used to invoke the Shell, e.g. -r

Parameters set as the Script executes commands:

$! The PID of the last background process

$? The return code from the last command executed

# Parameter Code Example

So let's put all of it into action in a Shell Script...

```
$ cat second.ksh
print $$
print $0
print "$# PPs as entered"
print "PP1=$1 PP2=$2 PP3=$3 PP4=$4"
shift
print $0
print "$# PPs after a shift"
print "PP1=$1 PP2=$2 PP3=$3 PP4=$4"
set "$@"
print 'Set "$@" - parameters in double quotes'
print "PP1=$1 PP2=$2 PP3=$3 PP4=$4"
set "$*"
print 'Set "$*" - parameters space separated'
print "PP1=$1 PP2=$2 PP3=$3 PP4=$4"

$ _
```

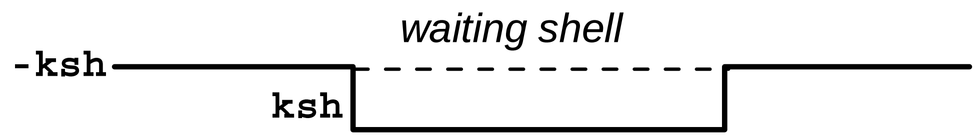
# Parameter Output Example

Here's what it does...

```
$ second.ksh arg1 arg2 "arg3 and text"
4687
second.ksh
3 PPs as entered
PP1=arg1 PP2=arg2 PP3=arg3 and text PP4=
second.ksh
2 PPs after a shift
PP1=arg2 PP2=arg3 and text PP3= PP4=
Set "$@" - parameters in double quotes
PP1=arg2 PP2=arg3 and text PP3= PP4=
Set $* - parameters in double quotes
PP1=arg2 arg3 and text PP2= PP3= PP4=
$ _
```

# This Shell and the Next

What happens to variables when you spawn a Sub-Shell?




Unless you <u>export</u> variables, they will not be passed on.

| | |
|---|---|
| `$ set` | *to list all variables and values* |
| `$ export var`<br>- or -<br>`$ typeset -x var` | *export variable var so that it will be inherited by Sub-Shells, or use typeset in the Korn Shell* |
| `$ export`<br>- or -<br>`$ typeset -x` | *to list variables that are exported, other variables will be unset in a Sub-Shell* |

# Inheritance Example

Let's see inheritance in action...

| | |
|---|---|
| `$ x=324` | *We can set a variable x* |
| `$ print "$$: X=$x"` | *in our current shell* |
| `4589: X=324` | |
| `$ ksh` | *In a Sub-Shell, x is unset* |
| `$ print "$$: X=$x"` | *- there is no value to print* |
| `4590: X=` | |
| `$ _` | |
| `Ctrl-d` | *Returning to the main Shell...* |
| `$ print "$$: X=$x"` | |
| `4589: X=324` | *x will have its value restored* |
| `$ export x` | *If we export x, a Sub-Shell* |
| `$ ksh` | *can inherit the value of x* |
| `$ print "$$: X=$x"` | |
| `4591: X=324` | |
| `$ x=3` | *If we change x from the* |
| `$ _` | *Sub-Shell, the change does* |
| `Ctrl-d` | *not affect the main Shell* |
| `$ print "$$: X=$x"` | |
| `4589: X=324` | |

# Korn Shell Variables

Korn Shell sets certain variables each time they are referenced:

| | |
|---|---|
| `SECONDS` | seconds since Shell invocation |
| `RANDOM` | random number in the range 0 to 32767 |
| `LINENO` | current line number within a Shell Script or function |
| `ERRNO` | system error number of the last failed system call – a system-dependent value! |



# Environment Variables

Several variables define the environment of a Shell:

| | |
|---|---|
| `CDPATH` | a search path for the cd command |
| `HOME` | your home directory |
| `IFS` | input field separators (defaults to: space, tab, newline) |
| `MAIL` | the name of your mail file |
| `MAILCHECK` | mail check frequency (default 600 seconds) |
| `MAILMSG` | the "you have new mail" message |
| `PATH` | the system command search path |
| `PS1` | the primary Shell command prompt |
| `PS2` | a secondary prompt for multi-line entry |
| `SHELL` | the pathname of the Shell |
| `TERM` | the terminal type (selects terminfo file) |



# Korn Environment Variables

Korn Shell specific features require environment variables:

| | |
|---|---|
| `COLUMNS` | screen width |
| `EDITOR` | the editor for command line editing |
| `ENV` | program/script to be sourced for each new Shell |
| `FCEDIT` | an editor for the `fc` command |
| `FPATH` | a search path for function definition files |
| `HISTFILE` | your history file |
| `HISTSIZE` | limit of history commands accessible |
| `LC_COLLATE` | sorting sequence for pattern ranges |
| `LINES` | screen length |
| `OLDPWD` | previous working directory for `cd -` |

# Korn Environment Variables (Cont.)

| | |
|---|---|
| `OPTARG` | required value for an option – `getopts` |
| `OPTIND` | index of the next argument for `getopts` to process |
| `PPID` | the parent process id |
| `PS3` | prompt for the `select` command |
| `PS4` | debug prompt for ksh with the -x option |
| `PWD` | the current working directory |
| `REPLY` | set by `select` command and the `read` command if no argument is given |
| `TMOUT` | seconds to Shell timeout |
| `VISUAL` | a visual editor – overrides `EDITOR` |

# Summary

- Setting variables
- Referencing variables
- Positional Parameters
- Shifting arguments
- Setting Positional Parameters
- Shell parameters
- Inheritance
- Shell variables
- Environment variables

# Unit 3. Return Codes and Traps

# Objectives

In this unit we will learn about:

- Return values
- Exit Codes
- Conditional execution
- The test command
- Compound expressions
- File test operators
- Numerical expressions
- String expressions
- Korn Shell test operators
- Korn Shell [[ ]] expressions
- Signals
- Sending signals
- Catching signals



# Return Values

Each command, pipeline or group of commands returns a value to its parent process

**$?** contains the value of the return code

- **zero** means success

- **non-zero** means an error occurred

The single value returned by a pipeline is the return code of the last command in the pipeline

For grouped commands – that is, ( ) or { } – the return code is that of the last command executed in the group

# Exit Status

A Shell script provides a return code using the *exit* command

| | |
|---|---|
| `$ print $$` | *check the Shell process id* |
| `879` | |
| `$ ksh` | *start a new Sub-Shell* |
| `$ print $$` | *and check its process id* |
| `880` | |
| `$ exit` | *quit the Sub-Shell* |
| `$ print $?` | *and print the return code* |
| `0` | |
| `$ print $$` | |
| `879` | |
| `$ ksh` | *begin another Sub-Shell* |
| `$ print $$` | |
| `890` | |
| `$ exit 101` | *exit with a value to set* |
| `$ print $?` | *the return code* |
| `101` | |
| `$ print $$` | |
| `879` | |
| `$ _` | |

# Conditional Execution

A return code (or exit status) can be used to determine whether or not to execute the next command

- if command1 is successful execute command2

```
command1 && command2
```

```
$ rm -f file1 && print file1 removed
```

- if command1 is not successful execute command2

```
command1 || command2
```

```
$ who|grep marty || print Marty logged off
```

# The test Command

The test command is used for expression evaluation

```
test expression
```

- or -

```
[ expression ]
```

- returns zero if the expression is true
- returns non-zero if the expression is false

The Korn Shell provides an improved version

```
[[ expression ]]
```

- easier syntax
- includes same functionality as *test*
- additional operators
- Shell expansions prevented

# Compound Expressions

For the [ ] or test command

| | |
|---|---|
| `exp1 -a exp2` | binary and operation |
| `exp1 -o exp2` | binary or operation |
| `! exp` | logical negation |
| `\(  \)` | to group expressions |

For the [[  ]] syntax

| | |
|---|---|
| `exp1 && exp2` | true if both expressions are true - the second is only evaluated if the first is true |
| `exp1 \|\| exp2` | true if either expression is true - the second is only evaluated if the first is false |
| `! exp` | logical negation |
| `(  )` | to group expressions |

# File Test Operators

File status can be examined using several operators

| *Operator:* | *True if ...:* |
|---|---|
| `-s file` | file has a size greater than zero |
| `-r file` | file exists and is readable |
| `-w file` | file exists and is writable |
| `-x file` | file exists and is executable |
| `-u file` | file exists and has the SUID bit set |
| `-g file` | file exists and has the SGID bit set |
| `-k file` | file exists and has the SVTX sticky bit set |
| `-e file` | file exists |
| `-f file` | file exists and is an ordinary file |
| `-d file` | file exists and is a directory |
| `-c file` | file exists as a character special file |
| `-b file` | file exists as a block special file |
| `-p file` | file exists and is a named pipe file |
| `-L file` | file exists and is a symbolic link |

# Numeric Expressions

For arithmetic expressions and integer values use

| *Expression:* | *True if ....:* |
|---|---|
| `exp1 -eq exp2` | `exp1 is equal to exp2` |
| `exp1 -ne exp2` | `exp1 is not equal to exp2` |
| `exp1 -lt exp2` | `exp1 is less than exp2` |
| `exp1 -le exp2` | `exp1 is less than or equal to exp2` |
| `exp1 -gt exp2` | `exp1 is greater than exp2` |
| `exp1 -ge exp2` | `exp1 is greater than or equal to exp2` |

AU232314

# String Expressions

To examine strings use one of the following

| *Expression:* | *True if ....:* |
| --- | --- |
| `-n str` | `str is non-zero in length` |
| `-z str` | `str is zero in length` |
| `str1 = str2` | `str1 is the same as str2` |
| `str1 != str2` | `str1 is not the same as str2` |

# Korn Shell Test Operators

The Korn Shell provides a number of additional `test` operators

| *Expression:* | *True if ...:* |
|---|---|
| `file1 -ef file2` | file1 is another name for file2 |
| `file1 -nt file2` | file1 is newer than file2 |
| `file1 -ot file2` | file1 is older than file2 |
| `-O file` | file exists and its owner is the effective user id |
| `-G file` | file exists and its group is the effective group id |
| `-S file` | file exists as a socket special file |
| `-t des` | file descriptor *des* is open and associated with a terminal device |

# Korn Shell [[ ]] Expressions

When using the Korn Shell [[ ]] syntax there are a few extra expressions...

| *Expression:* | *True if ....:* |
|---|---|
| `str = pattern` | str matches pattern |
| `str != pattern` | str does not match pattern |
| `str1 < str2` | str1 is before str2 in the ASCII collation sequence |
| `str1 > str2` | str1 is after str2 in ASCII collation |
| `-o opt` | option *opt* is on for this shell |

You may use Shell metacharacters in the patterns

# Practice Test

```
$ [[ -s /etc/passwd || -r /etc/group ]]
$ print $?
```
***True or False?***

```
$ test -f /etc/motd -a ! -d /home
$ print $?
```
***True or False?***

```
$ x="005"
$ y=" 10"
$ test "$y" -eq 10
$ print $?
```
***True or False?***

```
$ [ "$x" = 5  ]
$ print $?
```
***True or False?***

```
$ [[  -n "$x"  ]]
$ print $?
```
***True or False?***

```
$ test  -S  /dev/tty0
$ print $?
```
***True or False?***

```
$ [[  1234 = +([0-9])  ]]
$ print $?
```
***True or False?***

# Signals

The kernel sends ***signals*** to processes during their execution

- certain system events issue signals when they
  - run out of paging space
  - receive special key sequences like <Ctrl-c>
- The **kill** command sends a specific signal to a process

# What You Can Do with Signals

Signals sent to processes may be

| | |
|---|---|
| • Caught | the process deals with it |
| • Ignored | nothing happens |
| • Defaulted | use default *handlers* |

# The Kill Command

- To send a signal to a process:

`kill -sig pid` -or- `kill -s sig pid`

- To signal the current process group:

`kill -sig 0` -or- `kill -s sig 0`

- To send a signal to all of your processes, except those with PPID 1 (do not use if you are root):
- 

`kill -sig -1` -or- `kill -s sig -1`

- To list all defined signals

`kill -l`

- To list the signal that caused an exit error

`kill -l $?`




AU232326

# Signal List

Here is a list of some useful signals

| Signal: | | Event: |
|---|---|---|
| 0 | EXIT | issued when a process or function completes (Shell specific) |
| 1 | HUP | you logged out while the process was still running – sent to Sub-Shells too |
| 2 | INT | interupt pressed (Ctrl-c) |
| 3 | QUIT | quit key sequence pressed (Ctrl-\) |
| 15 | TERM | default kill command signal |
| 18 | TSTP | process suspend (Ctrl-z) |

# Signal List (Cont.)

| ***Signal:*** | | ***Event:*** |
|---|---|---|
| 19 | CONT | continue if stopped – issued by **`kill`** to a suspended process before TERM or HUP |
| 29 | PWR | power failure imminent – save data now! |
| 33 | DANGER | paging space low |
| 63 | SAK | you pressed <Ctrl-x> and <Ctrl-r> the SAK sequence |

# Catching Signals with Traps

The `trap` command specifies any special processing you want to do when the process receives a signal:

To process signals

```
$ trap 'rm /tmp/$$; print signal!; exit 2' 2 3
```

To ignore signals

```
$ trap '' INT QUIT
```

To reset signal processing

`$ trap - INT QUIT` - or - `trap 2 3`

To list traps set

```
$ trap
```

# Trap Example

```
#!/usr/bin/ksh
# ps_monitor
# monitor processes using ps -elf at intervals
# of 30 seconds for 2 minutes.  If interrupted,
# a summary report is produced by executing
# psummary.
#
trap 'print $0: interrupt received ;
        ./psummary ;
        exit' 2 3 15
ps -elf > /tmp/pdata
sleep 30
ps -elf >> /tmp/pdata
sleep 30
ps -elf >> /tmp/pdata
sleep 30
ps -elf >> /tmp/pdata
sleep 30
ps -elf >> /tmp/pdata
trap - 2 3 15
```

# Summary

- Return values
- Exit status
- Conditional execution
- The *test* command
- Compound expressions
- File *test* operators
- Numerical expressions
- String expressions
- Korn Shell *test* operators
- Korn Shell *[[ ]]* expressions
- Signals
- Sending signals – *kill* command
- Catching signals – *trap* command

# Unit 4.  Flow Control

# Objectives

For practical Shell Scripts we need program logic:

- The *if - then - else* construct
- Conditional loops with *until* and *while*
- Specific value iteration with *for*
- Multiple choice pattern matching with *case*
- The *select* command for menus
- Breaking and continuing loops
- Doing nothing – the null command

# The *if - then - else* Construct

```
if expression1
then
    commands to be executed if
    expression1 is true
elif expression2
then
    commands to be executed if
    expression1 is false, and
    expression2 is true
elif expression3
then
    commands to be executed if
    expression1 and expression2
    are false, but expression3 is true
else
    commands to be executed if all
    expressions are false
fi
```




AU232402

# if Example

Here is a simple if construct:

```
#!/usr/bin/ksh
# Usage: goodbye username
#
if [[ $# -ne 1 ]]
then
        print  "Usage is:  goodbye username"
        print  "Please try again."
        exit 1
fi
rmuser $1
print "O.K., $1 is removed."
```

When we run "goodbye", this is what we get ...

```
$  goodbye
Usage is:  goodbye username
Please try again.
$  goodbye pete
O.K., pete is removed.
$ _
```

# Conditional Loop Syntax

```
until  expression
do
        commands executed
        when expression is false
done            #  optional  <  file
```

```
while  expression
do
        commands executed
        when expression is true
done            #  optional  <  file
```

# until Loop Example

The C compiler returns a non-zero exit code **until** its compilation is successful:

```
$ until  cc prog.c
> do
>   vi prog.c
> done
$ _
```

# while true Example

The Script "forever" is a tough cookie!

```
#!/usr/bin/ksh
# An endless loop with a trap for INT QUIT TSTP
trap 'print "hasta la vista - baby!"' 2 3 18
while true
do
        print  "I'll be back."
        sleep 10
done
```

| | |
|---|---|
| `$  forever` | |
| `I'll be back.` | *every ten seconds* |
| `I'll be back.` | *the script speaks!* |
| `I'll be back.` | |
| *`Ctrl-c`* | *an attempt to stop it...* |
| `hasta la vista - baby!` | *invokes the trap, and* |
| `I'll be back.` | *it carries on.* |
| `I'll be back.` | |

# for Loop Syntax

```
for  identifier  in  word1  word2  ...
do
      commands  using  $identifier
      more  commands
done
```

```
for  identifier
# equivalent to: for identifier in "$@"
do
      commands  using  $identifier which takes
      values from the positional parameters
done
```





AU232412

# for - in Loop Example

Here we have a quick tidy-up to delete files:

```
$  for  file  in  *.tmp
>  do
>   rm -f $file
>  done
$  _
```

Why use the option -f ?

# for Loop Example

The sample Script "getprice.ksh" will look up the price list:

```
#!/usr/bin/ksh
# getprice.ksh - select price from "pricelist" file
# for each item entered on the command line
# Usage: getprice item1 item2 ...
#
for item
do
        grep -i "$item" pricelist
done
```

```
$ getprice.ksh  "Shock Absorbers"  "Air Filter"
Front Shock Absorbers      49.99
Rear Shock Absorbers       59.99
Air Filter                 10.99
$ _
```

# The case Statement

**case** word **in**
( **pattern1** | *pattern2* | ... )
*action* **;;**
(*) *default* **;;**
**esac**

**case** $identifier **in**
(**pattern1**) command1
more_commands **;;**
*(pattern2 | pattern3)* *commands* **;;**
(*) *commands* **;;**
**esac**

# case Code Example

A guessing game of sorts:

```
#!/usr/bin/ksh
# Usage:  match string
# To see how lucky you are feeling today

case "$1" in
     Ace  )         print "You are really close."  ;;
     King  )        print "Missed it by that much."  ;;
     Queen  )       print "Finally!" ;;
     Jack  )        print "I hope you'll get it next time." ;;
     *  )           print "Guess again." ;;

esac
```

# Case Code Output

A casino dealer in the making?

```
$ match Three
Guess again.

$ match Jack
I hope you'll get it next time.

$ match Ace
You are really close.

$ match King
Missed it by that much.

$ match Queen
Finally!
```

# Mini Quiz

1. There can be any number of *elif* statements in an *if* – *then* – *else* construct.
2. *while - true* and *until - false* **—** are they equals or opposites?
3. The statement: "*for identifier* " takes its input from positional parameters.

# The Korn Shell select Syntax

**select** identifier **in** word1 word2 ...
**do**
commands using $identifier usually containing a *case* statement
**done**

---

**select** identifier
# equivalent to: **select** identifier **in** "$@"
**do**
commands using $identifier from positional parameters usually containing a *case* statement
**done**

# select Code Example

To help identify animals we have a "barn.ksh" Shell Script:

```
#!/usr/bin/ksh
# usage: barn.ksh
PS3="Pick an animal: "
select  animal  in  cow  pig  dog  quit
do
        case $animal in
        (cow)     print "Moo"
                ;;
        (pig)     print "Oink"
                ;;
        (dog)      print "Woof"
                ;;
        (quit)      exit
                ;;
        ('')    print "Not in the barn"
                ;;
        esac
done
```

# select Output Example

Running "barn.ksh" we can choose an animal to examine ...

```
$ barn.ksh

1) cow
2) pig
3) dog
4) quit

Pick an animal: 1
Moo
Pick an animal: 2
Oink
Pick an animal: 3
Woof
Pick an animal: 8
Not in the barn
Pick an animal: 4
$
```

# exit The Loop

In the Korn Shell script /usr/sbin/snap

```
...
if [ "$badargs" = n ]
then
  for choice in $cmplist
  do
  if [ "$component" = "$choice" ]
  then found=y ; break ;
  fi
  done
  if [ "$found" = y ]
  then
    if [ -r "$destdir/$component/$component.snap" ]
    then
    more $destdir/$component/$component.snap
    else
    echo "^Gsnap:  $destdir/$component/$component.snap not found"
    exit 25
    fi
  fi
else
    usage
    exit 26
fi
...
```

# break The Loop

The break command jumps out of **do . . . done** loops:

- exits from the smallest enclosing loop
- jumps out a specified *number* of layers/loops

**break** ***number***

```
select  choice  in  Backup  Restore  Quit
do
  case $choice in
  (Backup)  find . -print|backup -iqf /dev/rfd0
  ;;
  (Restore) restore -xqf /dev/rfd0
  ;;
  (Quit)    break
  ;;
  ('')      print "What ?" 1>&2
  ;;
  esac
done
```



# continue The Loop

The **continue** command begins the next iteration of a **do . . . done** loop:

- starts at the top of the smallest enclosing loop
- begins again a specified *number* of layers/loops out

**continue** ***number***

```
$ for File in *
> do
> if [[ -d $File ]]
> then
>       continue
> fi
> file $File
> done
$ _
```

# null Logic

Sometimes you require a command, but you don't actually want to do anything – a NULL command

```
:                    # a COLON character
```

```
sys_call parameter1 parameter2
if  [[  $?  -eq  0  ]]
then
        # Debug slot
        :
else
        print $0: Error: command failed
        exit $ERRNO
fi
```

***without the null command ":" this would be illegal syntax***

# Program Logic Constructs Example

Here's a Script to delete empty files:

```
#!/usr/bin/ksh
# Usage: delfile file1 file2 ...
while [[ $# -gt 0 ]]
do
        if [[ -f "$1" ]]
        then
                if [[ ! -s "$1" ]]
                then
                        rm $1 && print $1 deleted
                else
                        print $1 not deleted 1>&2
                fi
        elif [[ -d "$1" ]]
        then
                print $1 is a directory
        else
                print "$1" is a special file
        fi
        shift
done
```

# Summary

- The *if – then – else* construct
- Conditional loops with *until* and *while*
- Specific value iteration with *for*
- Multiple choice pattern matching with *case*
- The *select* command for menus
- Leaving loops – *Exit* and *Break*
- Begining again – *Continue*
- Doing nothing — the null command – **:**

# Unit 5. Shell Commands

# Objectives

We shall learn in this unit about some special built-in Shell commands:

- The Korn Shell print command
- Special printing characters
- The read command
- Option and argument processing with getopts
- Command line re-evaluation with eval
- History manipulations with fc
- The set command
- Shell options with set
- Shell invocation
- Built-in commands
- Shell commands provided by AIX



# The Print Command

The **print** command is the Korn Shell output mechanism:

| | |
|---|---|
| **print** argument ... | prints arguments to standard output separated by spaces |
| **print -** argument ... | to print arguments that look like options |
| **print -r** argument ... | RAW mode – do not interpret special characters |
| **print -R** argument ... | equivalent to "-" and "-r" |
| **print -n** argument ... | no trailing newline after output |
| **print -uN** argument ... | output sent to file descriptor **N** |
| **print -s** argument ... | output to the shell history file only |

# Special Print Characters

Backslash character sequences have special meaning (except in raw mode)

| | |
|---|---|
| **\a** | Alarm - ring the terminal bell |
| **\b** | Backspace |
| **\c** | Print without trailing newline (same as print -n) |
| **\f** | Form feed |
| **\n** | Newline |
| **\r** | Return |
| **\t** | Tab |
| **\v** | Vertical tab |
| **** | Backslash |
| **\0xxx** | Character with octal code `xxx` (up to three octal digits) |

# print Examples

When you use the **print** command, here's what you get...

```
$ print "Line 1\n\tLine2"
Line1
       Line2

$ print 'One quarter = \0274'
One quarter = ¼
$ print 'Backslash = \0134'
Backslash = \
$ print -r 'hi\\\\there 1'
hi\\\\there 1
$ print -r hi\\\\there 2
hi\\there 2
$ print 'hi\\\\there 3'
hi\\there 3
$ print hi\\\\there 4
hi\there 4
$ _
```

# The read Command

To get input while a Shell Script is running, use **read:**

```
read variable ...
```

The read command reads a line from its standard input

- Assigns input words to the variables
- Set remaining variables to null if too few words
- Set last variable to the remainder of the words if too few variables

For the Korn Shell, if no variables are specified, the **REPLY** variable is set to the whole input line



# read Examples

We can use **read** from the Shell prompt as well...

```
$ read var1 var2
123 456 789
$ print "var1 = $var1 \tvar2 = $var2"
var1 = 123        var2 = 456 789
$ read var1 var2
abc
$ print "var1 = $var1 \tvar2 = $var2"
var1 = abc        var2 =
$ read
hi there
$ print $REPLY
hi there
$ _
```

# read Command Options

The Korn Shell **read** command has some options:

| | |
|---|---|
| `read -r variable ...` | raw mode – \ is not taken as a line continuation character |
| `read -s variable ...` | record the input line in the history file and set variables |
| `read -uN variable ...` | read from file descriptor **N** |

You can specify a prompt for the command to display on standard error
Add a "?prompt" to the first variable

```
read  variable?prompt  variable ...
```

For example, to request a user for a text string:

```
read string?'Please enter a text string'
```

AU232512

# read Options Examples

```
#!/usr/bin/ksh
# Usage: readrun
# Prompt the user when asking for input.
read word1?"Enter some text : "  word2
print "Word1 = $word1 Word2 = $word2 \n"

$ readrun
Enter some text : The cursor appeared here
Word1 = The Word2 = cursor appeared here
$ _
#!/usr/bin/ksh
# Usage: readraw
# Read & print text_file in raw mode until EOF.
while read -r line
do
   print -R  "$line"
done  <  text_file
$ readraw
The first line of \ttext_file
-now the second
The last line of \ttext_file\t-\tend of file!\a
$ _
```

# Processing Options

Parameters on a script command line are of two types

- arguments – used in script
- options – used to tell the script things

General parameter/argument processing is difficult

Consider

```
$ myscript -a -f optionfile argfile
$ myscript -foptionfile -va argfile
```

Shell provides **getopts** as a solution

# The getopts Command

The **getopts** command processes options and associated arguments from a parameter list

```
getopts optionstring variable parameter...
```

- Each invocation of **getopts** processes the next option in the `parameter` list
  - usually called within a loop
- The `optionstring` lists expected option identifiers
  - if an option identifier requires an associated argument, add a colon (:)
  - a leading colon in the list suppresses "invalid option" messages by `getopts`

# getopts Syntax Example

How are options processed when passed to a script?

Assume

- The possible options are a, b and c
- Option b is to have an associated argument
- Suppress normal OpSys error messages

Inside the script **getopts** will be used early on:

```
while  getopts  ':ab:c'  flag arguments
do
```

*identify the values set by getopts*

```
done
```

A correct command line to the script might be

```
$ prog.ksh  +c  -ab  barg  --  arg1  arg2
```

What about?

```
$ prog.ksh  -c -b -a -- arg1 arg2
```

# getopts Example

```
#!/usr/bin/ksh
#  Example of getopts
USAGE="usage:  example.getopts.ksh [+-c] [+-v] [-a argument]"

while getopts :a:cv arguments
do
case $arguments in
     a)  argument=$OPTARG ;;
     c)  compile=on ;;
    +c)  compile=off ;;
     v)  verbose=on ;;
    +v)  verbose=off ;;
    : )  print "You forgot an argument for the switch called a."; exit ;;
    \?)  print "$OPTARG is not a valid switch" ; print "$USAGE" ; exit ;;
    esac
done

print "compile is $compile; verbose is $verbose; argument is $argument "

#END
```

# The eval Command

The Shell processes each command line read before invoking the relevant command(s).

If you want to re-read and process a command line, use **eval**:

- **Eval** processes its arguments as normal
- The arguments are formed into a space separated string
- The Shell then executes that string as a command line
- The return value is that of the executed command line



# eval Examples

Here are some eval command lines...

```
$ eval print '*sh'
getopts.example.ksh eval.ksh         try.sh
```

```
$ message1=Goodbye
$ message10=Hello
$ variable=message10
$ eval print '$'$variable
Hello
```

*print the message named by $variable*

```
$ print "ls | sort -r" > cmd_file
$ read -r line < cmd_file
$ eval "$line"
zfile
afile
```

*read a cmd_file line - run as a command*

```
$ cmd='ps -ef | grep tommy'
$ eval $cmd
...
$ _
```

*run a string command to list tommy's processes*

# The fc Command

The Korn Shell fc command interactively edits and then re-executes portions of your command history file:

| | |
|---|---|
| `fc start end` | edits and executes a command range<br>– *start* defaults to the last command<br>– *end* defaults to the value of *start* |
| `-e editor` | to specify an editor other than **$FCEDIT** - Shell default is /bin/ed |

To re-execute a single command with automatic editing:

```
fc -e - old=new
command
```

- *old=new* to swap string *old* with string *new*
- *command* to specify a command - default last

# fc Examples - Edit and Execute

Ranges may be strings, absolute or relative numbers...

| | |
|---|---|
| `$ fc` | *edit the last command with the $FCEDIT editor, and then re-execute* |
| `$ fc pwd cc` | *edit with $FCEDIT from the most recent command starting with pwd, to one beginning with cc* |
| `$ fc -e vi 10 20` | *use vi to edit history lines 10 to 20* |
| `$ fc -e ex -3 -1` | *edit the last three commands with ex* |

Automatic editing can specify a command in a similar way

| | |
|---|---|
| `$ fc -e -` | *re-execute last command as was* |
| `$ fc -e - cc` | *re-run most recent cc command* |
| `$ fc -e - 2=3 10` | *swap 3 for 2 in command 10* |
| `$ fc -e - s=\? -2` | *change "s" into "?" in the command before last* |

# fc Examples - Lists

The Korn Shell fc command lists portions of your command history file:

| | |
|---|---|
| **fc -l** start end | list the specified command range<br>- the default is the last 16 commands |
| **-n** | suppress command numbers in list |
| **-r** | reverses the order of commands |

For example...

| | |
|---|---|
| $ **fc -l** pg grep | *lists commands from the last pg to a grep* |
| $ **fc -l** 15 20 | *lists commands 15 to 20* |
| $ **fc -l** -5 -1 | *lists the last five commands* |

# The set Command

We have seen three functions performed by the **set** command:

| | |
|---|---|
| `set` | lists set variables with their values |
| `set value ...` | re-sets the positional parameters |
| `set -o vi` | enables Korn Shell line recall and editing |

This last form sets a Korn Shell option. There are several more options to set:

- Korn Shell options and settings are listed by **`set -o`**
- Turn option on with **`set -o option`** or **`set -L`** (where **`L`** is an option identifier)
- Turn option off using **`set +o option`** or **`set +L`**

# Shell Options With Set

| *Option:* | *L* | *Description:* |
|---|---|---|
| `allexport` | `a` | automatically export each variable set |
| `bgnice` | | run all background jobs at a lower priority – this is on by default for interactive Shells |
| `ignoreeof` | | stops an interactive Shell exiting on Ctrl-d – you must use the exit command |
| `noclobber` | `C` | stops the Shell overwriting existing files with > re-direction ( >\| works instead) |
| `noexec` | `n` | for a non-interactive Shell to check syntax without executing commands |
| `noglob` | `f` | disables metacharacter pathname expansion |

# Shell Options With Set (Cont.)

| *Option* | *L* | *Description* |
|---|---|---|
| `notify` | `b` | to notify asynchronously of background job completions |
| | `s` | to sort positional parameters |
| `trackall` | `h` | set-up a tracked alias for each new command – on for non-interactive Shells |
| `verbose` | `v` | to display input on standard error as it is read |
| `vi` | | turns on history line recall and **vi** editing |
| `xtrace` | `x` | the debug option – the Shell displays PS4 with each processed command line |
| `errexit` | `e` | exits if any command returns a non-zero return code |
| `nounset` | `u` | displays an error message when an unset variable is used |




AU232536

# Set Quiz

1. What command would you use to re-set the positional parameters to "one" "two" "three"?

2. What lists the Shell options with settings?

3. Which *set* option ensures that each variable assignment will be inherited by a sub-Shell?

4. What would stop <Ctrl-d> from logging me out?

5. How can I use *set* to protect my files from being overwritten by output re-direction?

# Shell builtin Commands

We have seen the following builtin Shell commands:

| | | | |
|---|---|---|---|
| <u>.</u> | <u>:</u> | *bg* | <u>break</u> |
| **cd** | <u>continue</u> | **echo** | <u>eval</u> |
| <u>exec</u> | <u>exit</u> | <u>export</u> | *fc* |
| *fg* | *getopts* | *jobs* | **kill** |
| *print* | **pwd** | **read** | <u>readonly</u> |
| **set** | <u>shift</u> | **test** | **[ ]** |
| <u>trap</u> | <u>typeset</u> | **unset** | **wait** |

In the later units we will see:

| | | | |
|---|---|---|---|
| *alias* | *command* | *let* or *(( ))* | <u>return</u> |
| <u>times</u> | **ulimit** | *unalias* | *whence* |

All builtin commands can run in the current environment

<u>Special builtin commands may terminate the Shell if an error occurs</u>

# AIX Shell Commands

Some built-in Korn Shell commands are also provided as AIX commands – accessible from all Shells:

| | | | |
|---|---|---|---|
| **`alias`** | **`bg`** | **`cd`** | **`command`** |
| **`echo`** | **`fc`** | **`fg`** | **`getopt`** |
| **`jobs`** | **`kill`** | **`newgrp`** | **`read`** |
| **`umask`** | **`unalias`** | **`wait`** | |

AIX commands are also provided for the logical words:

**`false`**     **`true`**

Most of these commands are shell scripts in /usr/bin – they are provided for POSIX compliance

# Summary

- The Korn Shell print command
- Special printing characters
- The *read* command
- Option and argument processing with *getopts*
- Command line re-evaluation with *eval*
- History manipulations with *fc*
- The *set* command
- Shell options with *set*
- Shell invocation
- Builtin commands
- Shell commands provided by AIX

AU232552

# Unit 6.  Arithmetic

# Objectives

In this unit we will learn how to do arithmetic in the Shell.

- The expr utility
- Expr arithmetic and logical operators
- Korn Shell let or ((  ))
- Number bases
- Let logical operators
- Integer variables
- Implicit let
- The bc utility

# expr Arithmetic

AIX provides the **expr** utility to perform ***integer*** arithmetic

```
expr  argument1  operator  argument2 …
```

*expr* features

- runs in a Sub-Shell – not a Shell builtin command
- writes results to standard output
- exit code is 0 for non-zero evaluations
- exit code is 1 for zero or null evaluations
- exit code is ≥ 2 if an expression is invalid

Mostly used for control flow in shell scripts – loop counters



# expr Arithmetic Operators

To group expressions use:

| | |
|---|---|
| `( )` | fixes evaluation order - otherwise normal rules of precedence apply |

The integer operators result in mathematical evaluations:

| | |
|---|---|
| `*` | multiplication |
| `/` | integer division |
| `%` | remainder |
| `+` | addition |
| `-` | subtraction (also unary minus sign) |

NOTE: Use of backslash?

# expr Logic Operators

For integers or strings the following result is 1 for true, 0 for false:

| | |
|---|---|
| `=` | equal |
| `!=` | not equal |
| `<` | less than |
| `<=` | less than or equal |
| `>` | greater than |
| `>=` | greater than or equal |

Logic operators & (and) and | (or) give different output:

| | |
|---|---|
| **`expr LHS \& RHS`** | "and" - results in LHS if both sides are non-zero, 0 otherwise |
| **`expr LHS \| RHS`** | "or" - evaluates to LHS if it is non-zero, otherwise to RHS |

# expr Examples

Here is some simple integer arithmetic...

```
$ var1=6; var2=3
$ expr  $var1  /  $var2
2
$ expr  $var1  -  $var2
3
$ expr  \(  $var1  +  $var2  \)  \*  5
45
$ _
```

What is the result of the following?

```
$ expr  10  %  3

$ expr  10  /  3
```

# expr Examples (Cont.)

Some logical examples...

```
$ expr  abc  \<  def
1                        meaning true with expr
$ expr  3  \>=  4
0                        meaning false
$ value=4
$ expr  5  !=  $value
1
$ _
```

What is the result of the following?

```
$ expr 10  \|   3

$ zero=0
$ expr 10 \&    1  + $zero
```

# The Korn Shell let Command

```
let argument ..
```

-or-

```
((  argument  ))
```

- The *let* built-in Shell command performs long integer arithmetic approximately 10 times faster than *expr*
- Evaluates each argument as an arithmetic expression
- No quotes for special characters, or arguments with spaces or tabs in them, within ((  ))
- Variables need no $
- The exit code is 0 (true) for non-zero, and 1 (false) for zero evaluations

AU232610

# let Arithmetic Operators

For simple arithmetic:

| | | |
|---|---|---|
| ( | ) | overrides normal precedence rules |
| | * | multiplication |
| | / | division |
| | % | remainder |
| | + | addition |
| | - | subtraction (or unary minus) |
| | = | assignment |

**`var op= exp`** means `var = var op exp`

Upto nine levels of nested processing will be evaluated:

```
$ z=2  ;  y="z + 1"
$ ((  x=3*y  ))
$ print  $x
9
$ _
```

AU232612

# base#number Syntax

With **let** you are not limited to just decimal (base ten) integers:

- **let** constants are of the form ***base*#number**
- ***base*** is an integer in the range 2 to 36 (10 default)
- **number** may include upper or lower case letters for bases greater than 10

| | | |
|---|---|---|
| 2#100 in binary | = | 4 in base 10 |
| 8#33 in octal | = | 27 |
| 16#b in hexadecimal | = | 11 |
| 16#2A in base16 | = | 42 |

# let Arithmetic Examples - 1

Some simple arithmetic...

```
$ a=1
$ b=2
$ ((  z = 2#10 + -b ))        unary minus needs a space before it
$ let  c=a+b  d=b\*b          no spaces, but \ needed for *
                              multiple arguments
$ (( e  =  9 / 2#10 ))        integer division
$ (( e  +=  a ))              assignment: addition
$ print $z $a $b $c $d $e
```

What do you think we get?

# let Logical Operators

Logical expressions evaluate to 1 if true, 0 if false
(the exit code is 0 for non-zero, 1 for zero – as expected):

| | |
|---|---|
| `!` | logical negation |
| `<` | less than |
| `<=` | less than or equal to |
| `>` | greater than |
| `>=` | greater than or equal to |
| `==` | equal to |
| `!=` | not equal to |
| `&&` | logical "and" = 1 if both LHS and RHS are true (RHS not evaluated if LHS is false) |
| `\|\|` | logical "or" = 1 if either LHS or RHS are true (if LHS is true, RHS not used) |

# let Logical Examples

```
$  (( p = 9 ))

$  (( p = p * 6 ))
$  print $p
54

$ (( p > 0 && p <= 10 ))
$ print $?
1

$ q=100
$ (( p < q || p == 5 ))
$ print $?
0

$ if (( p < q && p == 54 ))
> then
> print TRUE
> fi
TRUE

$ _
```

# Korn Shell integer Variables

Korn Shell variables are stored as character strings unless defined with the *integer* command

```
integer variable=value ...
```

-or-

```
typeset -iN variable=value ...
```

- Sets the **integer** attribute for each variable
- *typeset* can define a base ***N***, variables then print in the specified base (2 to 36)
- Assignment to an **integer** variable causes expression evaluation – an implicit *let* command
- **let** does not have to convert **integer** variables from character strings to numerical values

# integer Examples

Some examples of **integer** and **typeset -i**...

```
$ integer x                        x can hold only integers
$ x=string
ksh:  string: 0403-009 The specified number is
not valid for this command.
$ x=5+10                           implicit let command
$ print $x
15
$ (( x = 5 + 100 ))
$ print $x
105
$ typeset -i8 nums0 nums1 nums2
$ nums0=8#5                        define an octal integer variable
$ nums1=8#10
$ (( nums2=8#3*nums0 ))            assign value
$ print ${nums2}
8#17
$ x=${nums2}
$ print $x                         print gives answer in base 10
15
$ _
```

# Implicit let Command

**integer** variable assignments are an implicit *let* command

Other implicit let commands are:

- Values for the Korn Shell **shift** command

```
shift OPTIND-1
```

- Resource limits with **ulimit**

```
ulimit -t TMOUT+60
```

# bc - Mathematics

The AIX system provides the bc utility

```
bc  [file]
```

- performs floating point arithmetic
- acts as a filter command or interactively
- reads arithmetic expression strings from standard input or a specified file
- semicolons or new lines separate expressions
- set the **scale** variable inside **bc** to define the required number of decimal places
- prints results to standard output

# bc Operators

For simple arithmetic and logical evaluations, use:

| | |
|---|---|
| `(, ), +, -, *, /, %, =` | as for **let** arithmetic operators |
| `==, !=, <, <=, >, >=` | as for **let** logical operators |
| `x^y` | raise x to the power y |
| `sqrt (x)` | square root |
| `x++ ++x` | post and pre increment x |
| `x-- --x` | post and pre decrement x |
| `x op= y ≡ x = x op y` | for +=, -=, *=, /=, %=, ^= |

A library provides complex mathematical functions:

| | |
|---|---|
| `s(x)` | sine of x |
| `c(x)` | cosine of x |
| `e(x)` | natural exponential of x |
| `l(x)` | natural log of x |
| `a(x)` | arctangent of x |
| `j(n,x)` | Bessel function |

Precision functions:

| | | |
|---|---|---|
| `length(n)` | number of significant digits | E.g. 123.456 has n=6 |
| `scale(n)` | number of digits after decimal point | E.g. 123.456 has n=3 |

# bc Examples

Here are some examples of **bc** working both as a filter and interactively...

| | |
|---|---|
| `$ print '1/4' \| bc` | *integer division without a scale* |
| `0` | |
| `$ print 'scale = 3 ; 1/4' \| bc` | *explicit scale value set* |
| `0.250` | |
| `$ print '5.5 * 2.2' \| bc` | *scale set implicitly from input* |
| `12.1` | |
| `$ bc` | |
| `sqrt( 4 )` | *no prompt – this is my input* |
| `2` | *the result from the command* |
| `Ctrl-d` | *to end interactive mode* |
| `$ _` | |

# Summary

- The *expr* utility
- *Expr* arithmetic and logical operators
- Korn Shell *let* or *((  ))*
- Number bases
- *let* logical operators
- Integer variables
- Implicit *let*
- The *bc* utility

# Unit 7. Korn Shell Types, Commands and Shell Functions

# Objectives

This unit describes Korn Shell arrays and takes an in-depth look at commands and their use

- Korn Shell arrays
- Command substitution
- Functions
- Typeset command
- Autoload functions
- Command aliases
- Pre-set aliases
- Tracked aliases
- The whence command
- Command line processing

# Defining Arrays

The Korn Shell supports one-dimensional arrays:

- arrays need not be "declared"
- access an element of an array by a subscript to a variable name
- any variable with a valid subscript becomes an array
- a subscript is an expression enclosed within [ ]
- subscripts should lie in the range 0 to 4095
- variable attributes (e.g. **readonly**) apply to all elements of the array

**Caution:** an entire array cannot be exported, only the 0th element

# Assigning Array Elements

Just like ordinary variables, values can be assigned, and later referred to:

- assign contents to an array element using

  **array[N]=*argument***

- to **unset** an array and assign new values sequentially, use

  **set -A array *argument* ...**

- to simply replace existing array values with new ones, use

  **set +A array *argument* ...**



# Referencing Array Elements

The $ notation is used to refer to the value in a variable:

- when referencing an array element use { } notation

```
print  ${array[N]}
```

- to refer to all the elements of an array use an `*` or `@` subscript (to give a space separated list)

`${array[*]}` or `${array[@]}`

- if you omit a subscript, it means the zeroth element

```
${array[0]}    ==    $array
```

# Array Examples

| | |
|---|---|
| `$ list[0]="Line 0"`<br>`$ list[1]="Line 1"`<br>`$ list[3]="Line 3"` | *Fill the array list.* |
| `$ print $list`<br>`Line 0` | *Print the zeroth element.* |
| `$ print ${list[*]}`<br>`Line 0 Line 1 Line 3` | *Print all elements.* |
| `$ print ${list[0]}`<br>`Line 0`<br>`$ print ${list[1]}`<br>`Line 1` | *Print elements individually.* |
| `$ print ${list[2]}` | *Element [2] is null.* |
| `$ print ${list[3]}`<br>`Line 3` | |
| `$ print $list[1]`<br>`Line 0[1]`<br>`$ _` | *Without { } notation, we get "$list" + "[1]".* |

# Another Array Example

Here we have the beginnings of a card game...

```
#!/usr/bin/ksh
# Usage: pickacard.ksh
# To choose a random card from a new deck
integer number=0
for suit in CLUBS DIAMONDS HEARTS SPADES
  do
  for n in ACE 2 3 4 5 6 7 8 9 10 JACK QUEEN KING
    do
      card[number]="$n of $suit"
      number=number+1
    done
  done
print  ${card[RANDOM%52]}

$ pickacard.ksh
QUEEN of DIAMONDS
$ _
```

# Command Substitution

Command substitution allows you to use the output of a command or group of commands:

- in a variable assignment
- in part of an argument list

| | |
|---|---|
| ***Bourne*** | `variable=```command`` |
| | - or - |
| ***Korn*** | `variable=$(command)` |

Nesting is possible:

```
var=`cmd1 \`cmd2 \\\`cmd3\\\` \` `
```

- or -

```
var=$(cmd1 $(cmd2 $(cmd3) ) )
```

# Command Substitution Examples

Here is command substitution in action...

```
$ d=$(date)
$ print  $d
Tue Feb 29 02:29:00 EST 2000
$ print "Contents of a file" > tmp_file
$ c=`cat tmp_file`
$ r=$(< tmp_file)                    no command, no Sub-Shell, so faster
$ print  "Cat: $c \n<: $r"
Cat: Contents of a file
<: Contents of a file
$ print "Most recent file: $(ls -t | head -1)"
Most recent file: tmp_file
$ arg1=1  ;  arg2=2
$ answer=$(expr $arg2 \* $(expr $arg1 + 3) )
$ print $answer
8
$ _
```

# Defining Functions

Commands can group together and be named
The set of commands form the function body
Function definitions look like:

| ***Bourne*** | ***Korn*** |
|---|---|
| `identifier()`<br>`{`<br>   *`commands`*<br>`}` | `function identifier`<br>`{`<br>   *`commands`*<br>`}` |

Functions

- provide a means of breaking down programs into discrete units
- stored in memory for fast access
- executed, like new commands, in the current environment




AU232716

# Functions and Variables

Functions have different variables to the main Script:

- arguments
  - taken as positional parameters to the function
  - calling script **`$1-${n}`** parameters are reset on leaving the called function
- variables
  - declared with the **typeset** or **integer** commands (inside a Korn Shell function) are "local" variables to the function
  - all other variables are "global" in the Script
  - the "scope" of a "local" variable includes all functions called from the current function

# function Examples

Some useful functions...

```
$ function cd
> {
>     command cd "$@"          - command stops recursion
      PS1="`pwd` : "           - PS1 is set to "/tmp : "
> }
$ cd /tmp
/tmp : cd /                    - the new prompt appears
/ : _                          - and will follow us around
```

```
#  Handy for usage errors in Shell Scripts
#  Invoke function usage with arguments: script
#  followed by arglist.  Note exit status!
function usage
{
    prog="$1"; shift
    print -u2 "$prog: usage: $prog $@"
    exit  1
}
```

# Ending Functions

A function completes after executing the last command:

- the exit code is normally that of the last command

- **return** can be used to specify an exit code *N*, or just end the function at that point

```
return N
```

- **exit** will terminate the current function and current Shell

```
exit N
```

- errors within a Korn Shell function cause it to return control and the error exit code to the calling Script

Functions may be deleted from memory using...

```
unset -f functionname
```

# Functions and Traps

The behavior of **trap** with functions is determined by the Shell type:

| | |
|---|---|
| Bourne: | a **trap** is "global" – the same in and out of a function |
| Korn: | a trap is "local" to a function and is reset on completion |
| | a main program trap is not shared with functions |
| | a signal that is not caught or ignored, may cause the script to terminate |
| | a signal that is ignored by a Korn Shell, is also ignored by functions called from it |

AU232724

# The typeset Command

The Korn Shell typeset command defines or lists variables and their attributes:

```
typeset ±LN variable1=value1 variable2=value2 ...
```

omitting variables lists variables with specified attributes

**-** sets attributes, or lists names and values

**+** unsets attributes, or lists just names

Where **L** is any of ...

| | |
|---|---|
| `r` | the **readonly** attribute – no modification of variables' value |
| `i` | sets the **integer** attribute – use with *N* to set number base |
| `x` | the **export** attribute – the variable will be exported |

# typeset Examples

Declare arrays to specify:

- size
- attributes

```
$ typeset -xi8 a2[1]          exported & octal integer
$ a2=52
$ a2[1]=25
$ ksh
$ print $a2 ${a2[1]}
8#64                          only element 0 was exported
$ _
```

Inside a Korn Shell function, **typeset** creates a "local" variable...

```
# Function to convert numbers into binary
function  binary_convert
{
      typeset -i2 binary=$1
      print "$1 = $binary"
}
```

# typeset With Functions

Other uses of **typeset** are:

- display functions
- set function attributes
- unset function attributes

*typeset ±fL function1 function2 ...*

- to list functions with specified attributes, omit function list
- ***-f*** sets attributes, or displays function names and definitions
- ***+f*** unsets attributes, or displays only function names

Where ***L*** is any of...

| | |
|---|---|
| *x* | the **export** attribute – the function will be available to implicit Shells invoked from the current one |
| *u* | to mark a function as undefined |
| *t* | the Shell **xtrace** option for a function |

# typeset with Functions Examples

```
$ typeset -f                    lists functions in full
function list
{
    while [[ "$1" != "X" ]]
    do
        print  $1
        shift  1
    done
}
$ typeset -fx list              export the list function
$ typeset +f                    lists function names
list
$ _
```

# autoload Functions

A Korn Shell function that is defined only when it is first called, is an **autoload** function:

```
autoload function

- or -

typeset -fu function
```

- using **autoload** functions improves performance
- the Korn Shell searches directories listed in the **FPATH** variable for a file with the name of the called function
- the contents of that file then defines the function
- existing function definitions are not unset




AU232734

# Aliases

The Korn Shell **alias** facility provides:

- a way of creating new commands
- a means of renaming existing commands

| | |
|---|---|
| Creation: | `alias name=definition` |
| Deletion: | `unalias name` |

An **alias** definition may contain any valid Shell Script or metacharacters

# Processing Aliases

Command lines are split into words by the Shell:

- check the first word of each command line for a defined **alias**
- a backslash in front of a command name prevents **alias** expansion if the alias exists
- if the **definition** ends in a space or tab, the next command word will also be processed for **alias** expansion
- **resolve alias** names within a function when function definitions are read – not at execution!

# Preset Aliases

Korn Shell uses the following exported aliases

- may be unaliased or redefined

```
alias autoload='typeset -fu'

alias false='let 0'

alias functions='typeset -f'

alias hash='alias -t'

alias history='fc -l'

alias integer='typeset -i'

alias nohup='nohup '          with trailing space

alias r='fc -e -'

alias true=:

alias type='whence -v'
```





AU232740

# The alias Command

The **alias** command has some options:

*`alias -L name=definition`*

Where ***L*** is any mix of...

| | |
|---|---|
| *`x`* | to set, or display exported aliases |
| *`t`* | to set, or list tracked aliases |

If ***definition*** is quoted...

| | |
|---|---|
| "definition" | interpreted when entered |
| 'definition' | text stored for later interpretation |




AU232742

# alias Examples

```
$ alias -x ls='ls -a'
```
*ls is set and exported*

```
$ x=10
$ alias px="print $x" rx='print $x'
$ x=100
$ px
10
```
*prints $x as it was*

```
$ rx
100
```
*prints the latest $x*

```
$ alias od=done
```
*an alias for some flow control*

```
$ for i in lazy done
> do
>       print $i
> od
lazy
done
```

# Tracked Aliases

A "tracked alias" reduces the search time for a future use of a command

```
set -o trackall  or  set -h
```

turns on Shell **trackall** option

First use of a command creates tracked alias

Force creation with

```
alias -t name
```

List all "tracked aliases"

```
alias -t
```

NOTE: the value of a "tracked alias" becomes undefined when the PATH variable is reset




AU232746

# The whence Command

**Whence** reports how a command will be carried out by the Korn Shell

*whence -pv command*

- **-v** for a verbose report
- **-p** to force a PATH search even if the command is an alias or function (AIX only option)

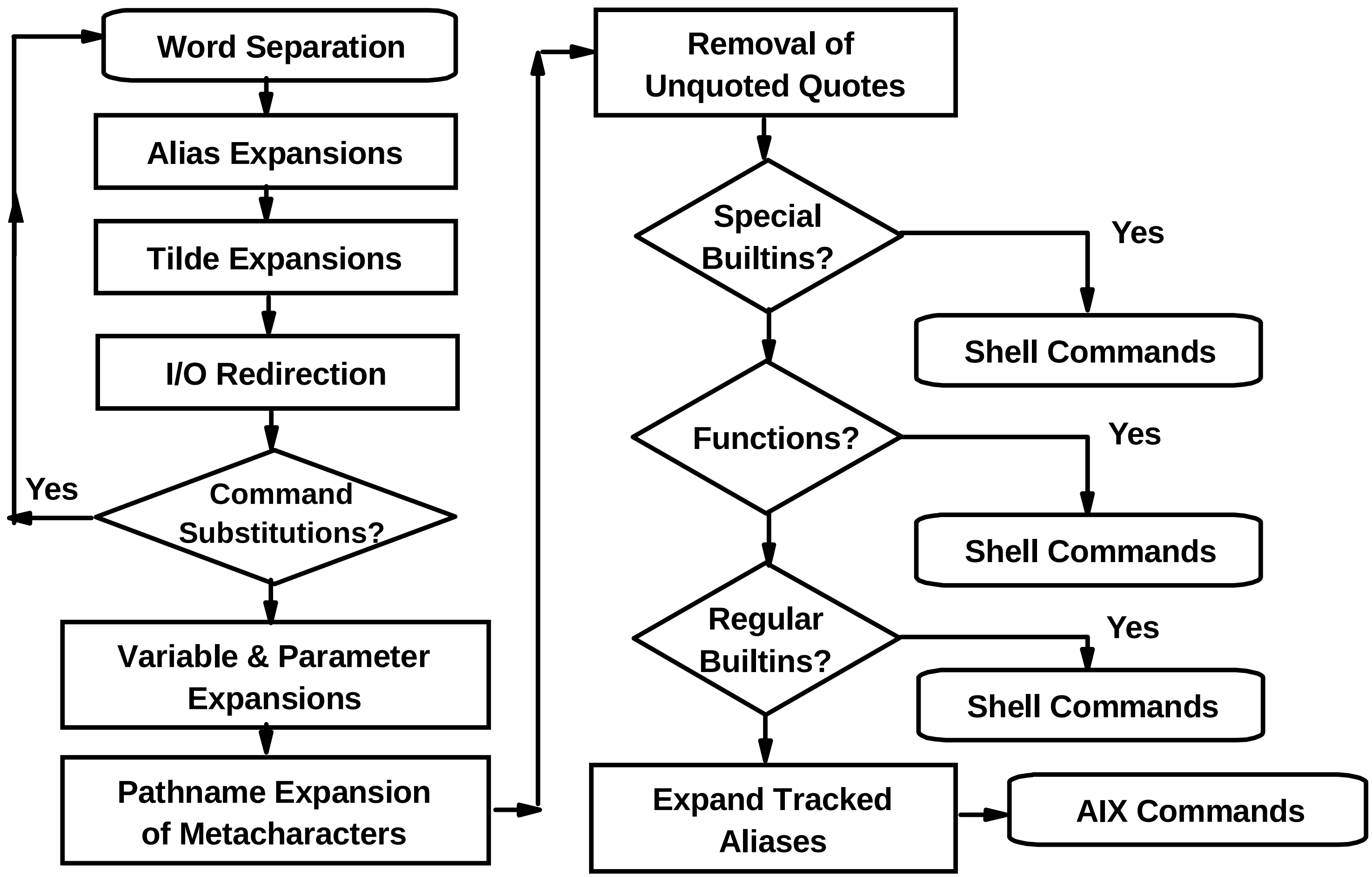
```
$ whence vi
/usr/bin/vi
$ whence -v vi                    executable program
vi is a tracked alias for /usr/bin/vi
$ whence -v print
print is a shell builtin
$ whence type                     so type is an alias
whence -v
$ type for
for is a reserved word
$ _
```




AU232748

# Command Line Processing

Each command line is processed in the following way by the Korn Shell:




AU232750

# Summary

- Korn Shell arrays – defining and referencing
- Command substitution
- Functions
- Typeset command
- Autoload functions
- Command aliases
- Preset aliases
- Tracked aliases
- The whence command
- Command line processing

# Unit 8. String Handling

# Objectives

This unit will show how to manipulate text (character) strings using Korn Shell variables:

- Variable replacements
- Variable sub-strings
- Variable lengths
- Further typeset options
- Tilde expansions

# Variable Replacements

Value of variables can be replaced with alternate values

| | |
|---|---|
| `${variable:-word}` | value is **word** if **variable** is unset (use default value) |
| `${variable:=word}` | value is **word** if **variable** is unset and assigns word to **variable** if it is unset (assign default value) |
| `${variable:+word}` | value is null if **variable** is unset, else value is **word** (use alternate value) |
| `${variable:?word}` | if **variable** is unset, **word** is displayed on standard error and the Shell script or function terminates with a non-zero exit code (exit 1) |

# Variable Replacement Examples

Some simple examples...

- To assign the value of TERM_DEF to TERM if it is unset or null:

```
TERM_DEF=ibm3162
...
print "TERM set as ${TERM:=$TERM_DEF}"
```

- Print date and time using command substitution, or what was set earlier (do not allow null date):

```
print  ${date:-$(date)}
```

- Using the alternate value "1" if variable has a value:

```
var_flag=${var:+1}
```

- To exit the script if positional parameter 3 was not given (it can be null):

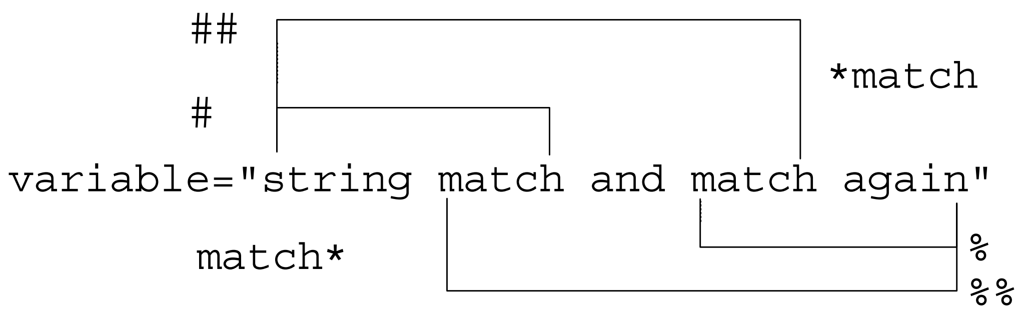
```
${3?"No parameter 3? exit"}
```

AU232804

# Korn Shell Sub-Strings

In the Korn Shell the ${ } syntax also works with patterns:

| | |
|---|---|
| `${variable#pattern}` | removes smallest matching left pattern from variable |
| `${variable##pattern}` | removes the largest matching left pattern |
| `${variable%pattern}` | removes the smallest right matching pattern |
| `${variable%%pattern}` | removes the largest matching right pattern |



# Korn Shell Sub-String Examples

A bit of chopping...

```
$ variable="Now is the time"
$ print ${variable#N*i}                  shortest left
s the time
$ print ${variable##N*i}                 longest left
me
$ print ${variable%time}                 shortest right
Now is the
$ print ${variable%%t*e}                 longest right
Now is
$ _
```

Here's a function to strip out the file name from its path and print it...

```
function base
{
     print ${1##*/}          # match what?
}
```

# Korn Shell Sub-String Quiz

Now it's your turn...

1. How can I strip the ".c" extension from a C program file name held in variable "name", and print it?

2. Write a function "path" to print the pathname part of a file name.

# Variable Lengths

A special Korn Shell variant of the **${}** syntax can be used to find the length of a variable:

- to find the number of characters in a variable...

```
${#variable}
```

- the number of positional parameters is...

```
${#*}          or          ${#@}
```

- for the number of elements set in an array (not the highest element subscript)...

```
${#array[*]}   or          ${#array[@]}
```

# typeset Options Review

**Typeset** command used to

- set attributes for variables or functions
- create local variables in functions

*typeset ±LN variable=value...*

where ***L*** is... *i* integer, ***N*** is a fixed base
*r* readonly
*x* to export the variable

*typeset ±fL function...*

where ***L*** is... *x* to export the function
*u* for an autoload function
*t* to set xtrace in the function

**-** to set attributes, display names and values

**+** to unset attributes or display just names

# Further typeset Options

Options below allow variables to be formatted upon expansion by the Korn Shell:

```
typeset ±LN variable=value...
```

where ***L*** is...

| | |
|---|---|
| *u* | convert ***value*** to uppercase when expanded |
| *l* | convert ***value*** to lowercase |
| *L* | left-justify, pad with trailing blanks to width ***N*** – if value is too big, truncate from the right |
| *R* | right-justify, adding leading blanks to width ***N*** – if wider than ***N***, truncate from the left |
| *LZ* | left-justify to width ***N*** and strip leading zeros |
| *RZ* | right-justify to width ***N***, adding lead zeros if the first character is a digit |

# typeset Examples

Here are the different types in action...

```
$ typeset -u var=upper
$ print $var
UPPER
$ typeset -l var=LOWER     # lower case ell
$ print $var
lower
$ typeset -L6 text=SIDE
$ print "${text}="
SIDE  =


$ typeset -R6 text
$ print "=$text"
=  SIDE


$ typeset -LZ4 num=000.1234567
$ print ${num}
.123
$ typeset -RZ5 num=1234567
$ print $num
34567
```

AU232818

# Tilde Expansions

Following alias expansion the Korn Shell checks for a leading unquoted ~ character to see if it is:

| | |
|---|---|
| ~ | tilde by itself is replaced by $HOME |
| ~+ | is replaced by $PWD |
| ~- | is replaced by $OLDPWD |
| ~user_name | is expanded into the $HOME value for the ***user_name*** given |
| ~other_text | will be left alone |

Examples...

```
cd ~            ≡    cd $HOME
lastdir=~-      ≡    lastdir=$OLDPWD
johns=~john     ≡    johns=/home/john
```

# Summary

- Variable replacements
  - *for unassigned/null strings*
- Variable sub-strings
  - *simple pattern matches*
- Variable lengths
  - *the # "operator"*
- Further typeset options
  - *justification and padding*
- Tilde expansions
  - *shortcuts*

# Unit 9. Regular Expressions and Text Selection Utilities

# Objectives

This unit will show how to select and manipulate text (character) strings using:

- Regular expressions
- The *grep* command
- The *tr* command
- The *cut* command
- The *paste* command

# Sample Data File

To manipulate data, we need to know its format.
The data file we will use in this unit has the following structure:

```
Lastname,<SPC>Firstname<TAB>nnn-mmmm

$ cat  phone.list

Terrell, Terry          617-7989
Franklin, Francis       704-3876
Patterson, Pat          614-6122
Robinson, Robin         411-3745
Christopher, Chris      305-5981
Martin, Marty           814-5587
Llewellyn, Lynn         316-6221
Jansen, Jan             903-3333
Llewellyn, Lee          817-8823
$ _
```

AU232902

# Regular Expressions

Powerful feature available in many programs

Used to **select** text

- vi, ex, emacs, grep/egrep, sed, awk, perl

What are they?

- An expression representing a pattern of characters
- Contain a sequence of characters/metacharacters

# Regular Expression Metacharacters

| Pattern | Meaning (matches) |
|---|---|
| aphanumeric character | The character itself (not really a metacharacter) |
| **.** (period) | Any single character |
| [AZ] | One of A or Z |
| [^AZ] | Any character not A or Z |
| [A-Z] | Any character in range A to Z |
| [-AZ] | One of -, A or Z |
| [0-9] | Any digit 0 to 9 |

# Extending the Pattern

Two ways:

- Anchors
- Multipliers

Anchors are

| | |
|---|---|
| ^ | Matches beginning of line |
| $ | Matches end of line |

Multipliers apply to patterns. They are

| | |
|---|---|
| * | zero or more occurences of previous pattern |
| ? | zero or one occurence of previous pattern |
| + | one or more occurences of previous pattern |
| {m,n} | at least m and no more than n occurences of previous pattern ("quoted braces") |

# Quoted Braces

To specify the number of consecutive occurences

**Syntax 1:** `regular_expression\{min, max\}`

To look for two, three or four occurrences of any combination of the characters 3, 4 and 5 consecutively

```
grep '[345]\{2,4\}' phone.list
```

**Syntax 2:** `regular_expression\{exact\}`

To look for any lines which have two consecutive "r" characters

```
grep 'r\{2\}' phone.list
```

**Syntax 3:** `regular_expression\{min,\}`

To look for any lines with at least two consecutive "r" characters preceded by an "e"

```
grep 'er\{2,\}' phone.list
```




AU232910

# Quoted Parentheses

To capture the result of a pattern

**Syntax**: `\(regular expression\)`

- Stores the character(s) that match the regular expression (within parentheses) in a register

- Nine registers are available; characters which match the first quoted parentheses are stored in register one, those that match the second quoted parentheses in register two, etc.

- To reference a register use a backslash followed by a register number:

    \1 to \9

For example, to list any lines in "phone.list" where there are two identical characters together...

```
grep '\(.\)\1' phone.list
```

# Regular Expressions – Quiz

Using the "phone.list" file, what RE gives:

1. People with five-letter surnames?
2. People with first names of at least four characters?
3. All entries where the number before the dash is the same as that after the dash e.g. 3-3456?
4. People whose surnames begin with A, B or C?

# grep Command

- Search file(s) or standard input for lines containing a match for a specific pattern

```
grep [options] pattern [ file1 file2 . . . ]
```

- Valid grep metacharacters: `. * ^ $ [-]`

| | |
|---|---|
| . | any single character |
| * | zero or more occurrences of the preceding character |
| ^a | any line that begins with "a" |
| z$ | any line that ends with "z" |
| [a-f] | any ONE of the characters in the stated range |

- Valid options:

| | |
|---|---|
| -c | print only a count of matching lines |
| -i | ignore the case of letters when making comparisons |
| -l | print only the names of the files with matching lines |
| -n | number the matching lines |
| -s | works silently, displays only error messages |
| -v | print lines that do NOT match |
| -w | do a whole word search |

# grep Examples

```
1.$ grep -i "tech support" phone.list

2.$ grep bob /etc/passwd

3.$ ps -ef | grep tracy

4.$ ls -l | grep '^d'

5.$ grep -n '.*' /etc/passwd > \
  > passwd.file.numbered.lines

6.$ egrep '^b(i|o)' /etc/passwd
```

# tr For Translations

The **tr** command translates one set of characters into another:

```
tr LISTIN LISTOUT < in_file > out_file

          - or -

tr -d LISTIN < in_file > out_file
```

- characters in **LISTIN** are replaced by the corresponding ones in **LISTOUT**
- if **LISTOUT** contains fewer characters than **LISTIN,** ignores extra ones from LISTIN
- if **LISTOUT** contains more characters than **LISTIN,** ignores extra ones from LISTOUT
- with **-d**, characters in **LISTIN** are deleted
- only works with STDIN and STDOUT

# tr Examples

Some simple translations...

```
$ print $HOME | tr "/" "-"
-home-team01
$ print "{ { [ ... ] } }" | tr "{}" "()"
( ( [ ... ] ) )
$ print "Lower to upper" | tr "[a-z]" "[A-Z]"
LOWER TO UPPER
$ print "TOP DOWN" | tr '[:upper:]' '[:lower:]'
top down
$ print "vowels and consonants" | tr -d aeiou
vwls nd cnsnnts
$ tr -d '\015' <dos_txt_file >aix_txt_file
$ _
```

# The cut Command

Cut extracts fields or columns from text input

```
cut -dS -s -flist [ file ]

        or

cut -cLIST [ file ]
```

**-dS** where S is the character to take as a delimiter

**-s** with -dS suppresses lines that do not contain delimiters

**-fLIST** specifies a **LIST** of fields to cut out and keep

**-cLIST** is a **LIST** of columns to cut (character positions)

**LIST**
- specifies field or column numbers
- may contain comma separated values (m,n) or a range (m-n)




AU232924

# cut Examples

Field numbering starts at 1

```
$ cut  -d:  -f1,3  /etc/passwd  |  head  -3
root:0
daemon:1
bin:2
$ cat  /etc/passwd  |  cut  -d'*'  -s  -f1
guest:
$ df  |  cut  -c6-10  |  tail  +2
hd4
hd2
hd3
hd1
$ text="A tasty dish to set before the King!"
$ echo  $text  |  cut  -c-8,32-
A tasty King!
$ _
```

# The paste Command

As name suggests, sticks (merges) things together

Commonly used to create or format a data stream

Default output is

line from file1 <TAB> line from file2

Separator(s) may be changed on command line

Options:

-d [dlist] the delimiter between files (may be a list)

-s make the output a single line of all lines of each file

# paste Examples

Print a 3 column listing of .ksh files:

```
ls *.ksh | paste - - -
```

Format a listing in 3 columns using <TAB> <TAB> <NEWLINE> as delimiters

```
ls *.ksh | paste -d"\t\t\n" -s -
```

# Summary

- Understand Regular Expressions
- Using the *grep* command to select text
- Using the *tr* command to translate characters
- Using the *cut* command to select text fields
- Using the *paste* command to merge data streams

# Unit 10. Utilities for Personal Productivity

# Objectives

This unit will introduce utilities that can improve your personal productivity – sed, tar, at, crontab

- use the stream edit utility – *sed*
- use the archive utility – *tar*
- manipulate when your work gets done – *at* and *crontab*

# sed


There are several ways of running sed:

- **sed** 'edit-instructions' filename
- command | **sed** 'edit-instructions'
- **sed** -f command.file filename

**Note:** The input file is not changed or overwritten by **sed**!

# Line Selection

The **sed** instructions operate on all lines of the input, unless you specify a ***SELECTION*** of lines:

```
sed 'SELECTION edit-instructions'
```

***SELECTION*** can be

- a single line number

| | |
|---|---|
| `1` | = line 1 of the input |
| `$` | = the last line of the input |

- a range of line numbers

| | |
|---|---|
| `5,$` | = from line 5 to the end of the input |

- a regular expression to select lines matching a pattern

| | |
|---|---|
| /string/ | = selects all lines containing "string" |

- a range using regular expressions

| | |
|---|---|
| /^on/,/off$/ | = from the first line beginning with "on" to the first ending in "off" |

# The Substitute Instruction

This instruction changes data

**Syntax:** `s/old string/new string/g`

Some examples

1. To replace the first occurrence of "Smith" on each line with "Smythe"

```
sed  's/Smith/Smythe/'  phone.list
```

2. To replace all occurrences of "Smith" with "Smythe" using a different delimiter

```
sed  's!Smith!Smythe!g'  phone.list
```

3. To precede each phone number with "Tel:"

```
sed  '/[0-9]\{3\}-[0-9]\{4\}/s//Tel: &/g' \
phone.list
```

# Substitutions - Quiz

1. Convert the "phone.list" into just a name list, i.e. get rid of the phone numbers

*output:*

```
Terrell, Terry
Franklin, Francis
Patterson, Pat
 ...,    ...
```

```
sed  's/____________________//'  phone.list
```

2. Convert the "phone.list" file to a first-name and number list

*output:*

```
Terry        617-7989
Francis      704-3876
Pat          614-6122
...             ...
```

```
sed  's/____________________//'  phone.list
```

# sed with Quoted Parentheses

- Repeating the first character

```
$ print "1234" | sed 's/^\(.\)/\1\1 /'
11234
$ _
```

*any single character to register 1*

*register 1 is repeated*

- Stripping out all but the first and last characters

```
$ print "1234"|sed 's/^\(.\).*\(.\)$/\1\2/'
14
$ _
```

*character to register 1*

*character to register 2*

*register 1 and 2*

Now it's your turn...

Working on the "phone.list" file, abbreviate everyone's first name to an initial and a period (use register 1 to store each initial)

```
sed  's/__________/__________/'  phone.list
```

# Summary for Substitutions

- without a "**g**", **sed** only substitutes the first match

```
$ print xxx | sed ' s/x/y/'
yxx
$ print xxx | sed 's/x/y/g'
yyy
$ _
```

- other delimiters can be used when "/" makes life difficult
  – e.g. converting an AIX to a DOS pathname

```
$ pwd | sed 's/\//\\/g'
\home\kim\desktop
$ pwd | sed 's;/;\\;g'
\home\kim\desktop
$ _
```

# Delete and Print

This command removes text

**Syntax:** SELECTION**d**

- To delete all lines in the output stream

```
$ sed  d  phone.list
```

- Delete from line 5 to the end of the file

```
$ sed  '5,$d'  phone.list
```

By default **sed** writes out every line it reads in

- makes print instruction "**p**" by itself redundant:

```
$ cat in.file
line 1
line 2
$ sed p in.file
line 1
line 1
line 2
line 2
$ _
```

# Append, Insert and Change

These instructions add or modify text

**Syntax**: SELECTION**x**
**text**

Where **x** is

i inserts **text** before a single selected line

a appends **text** after a matched line

c changes a range of matched lines into **text.**
SELECTION can be a single line or a range but only one copy of **text** is printed in its place

# Command Files

- A **sed** command file consists of one or more **sed** instructions on separate lines

- Command files are useful in many situations:
  - storing multiple instructions
  - storing a long complex command
  - for commands which may need to be modified and reused

- Use the "**-f**" option to use a command file

Example...

```
$ cat sedscript.sed
  s/ GA/, Georgia/
  s/ FL/, Florida/
  s/ IL/, Illinois/
  s/ TX/, Texas/
  s/ MD/, Maryland/
  s/ DC/, District of Columbia/
$ sed -f sedscript.sed addrs.file > new.addrs.file
$ _
```

AU232A18

# A Practical Example

Converting a "BookMaster" script to a "wysiwyg" file

```
:ul.
:li.An unordered list starts with ":ul.".
:li.Each list item is tagged with ":li." - it
appears as an indented bullet point.
:li.The end of the list is marked by ":eul."
:eul.
```

Strategy:

1. Remove lines which contain just ":ul." or ":eul."

2. For lines that start with ":li.", substitute the ":li." with a dash followed by five spaces

```
$ cat  bkm.wysi.sed
/^:e*ul\.$/d
s/^:li\./-     /
$ sed  -f bkm.wysi.sed  bookmaster.file > wysi.file
$ cat wysi.file
-     An unordered list starts with ":ul.".
-     Each list item is tagged with ":li," - it
appears as an indented bullet point.
-     The end of the list is marked by ":eul."
```

# Multiple Editing Instructions

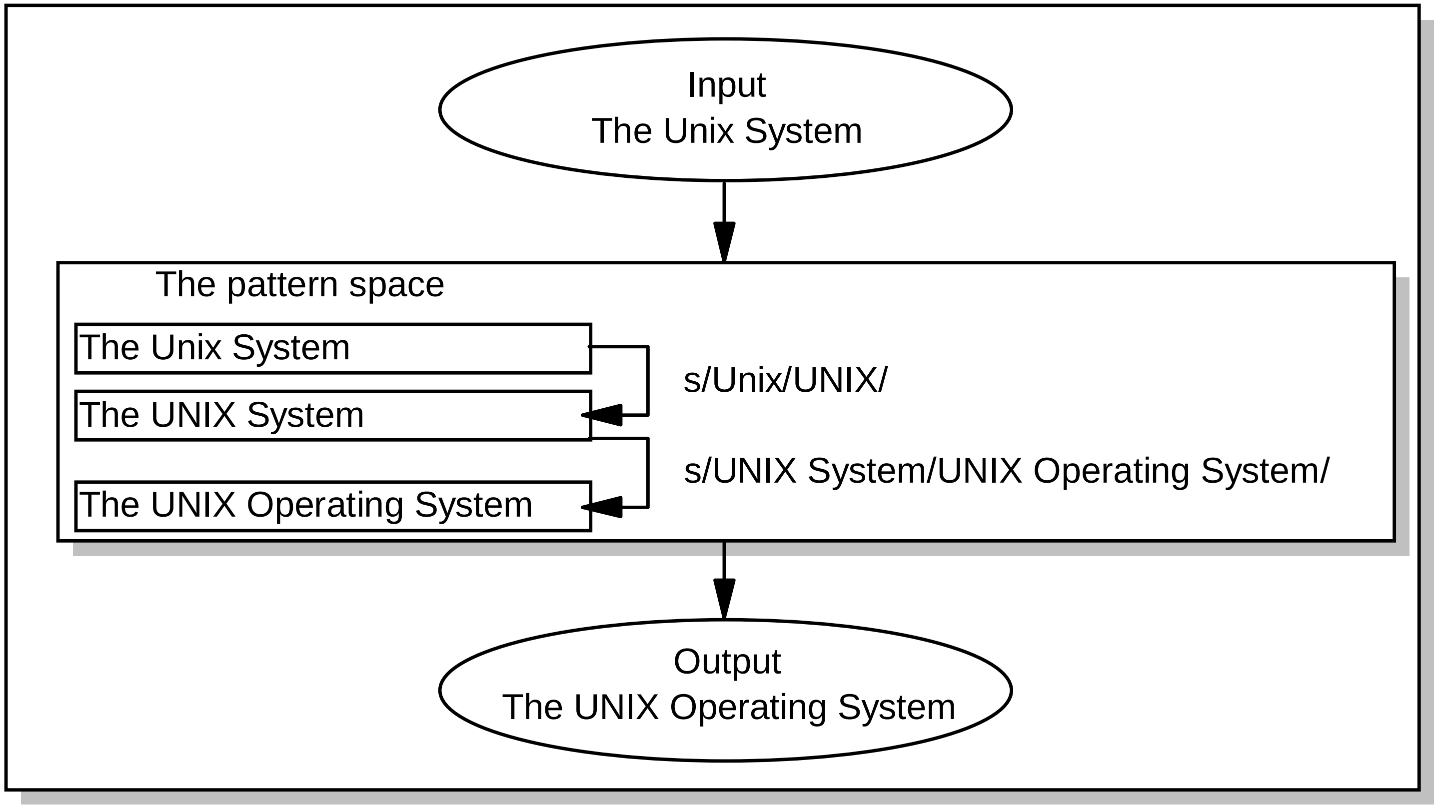
- Multiple instructions can be applied to each line
- Each instruction must be on a separate line

Example 1...

```
$ sed  '/[1-4]-/s/$/ (Bldng 1) /
>       /[5-9]-/s/$/ (Bldng 2) /'  phone.list

Terrell, Terry        617-7989  (Bldng 2)
Franklin, Francis     704-3876  (Bldng 1)
```

# Internal Operation




- sed applies all editing instructions to a line before it moves on to the next line
- it holds each input line in a "pattern space" or temporary buffer while editing instructions are applied in sequence

# Internal Operation – Example

Example of sed command/instructions

```
$ print "The Unix System" | sed  's/Unix/UNIX/
>         s/UNIX System/UNIX Operating System/'
The UNIX Operating System
$ _
```

# Grouping Instructions

Braces "{" "}" are used for two purposes:

- one *SELECTION* inside another (*nest*)
- to apply multiple instructions to the same *SELECTION* range (*group*)

Example...

*SELECTION range*

```
/\.ol/,/\.eol/{
                /^$/d
                }
```

*instructions for the SELECTION range*

- The instruction "/^$/d" (delete blank lines) will be applied to a range of lines between one that contains an ".ol" and up to the first containing an ".eol"
- The special meaning of the dot preceding "ol" and "eol" is escaped by the use of a backslash

# sed Advanced Topics

There are two other areas in *sed* that can be useful

- multiple input lines for the pattern space
- use of the hold space (temporary area)

There are three instructions for multiline input

- N Read next line
- P Print line
- D Delete line

Notice they are in UPPER CASE

# Multiple Input Lines - N Instruction

The **N** instruction

- does NOT clear pattern space
- inserts an (embedded) newline ("\n") into the pattern space
- reads a line and appends to the pattern space

Similar to **n** instruction

- BUT n clears pattern first

An embedded newline ("\n") can be matched explicitly

^ and $ refer to the FIRST and LAST character respectively of the pattern space

# The P and D Instructions

These also do not clear the pattern space

P prints the pattern space up to the first embedded newline

D deletes the text up to the first embedded newline

- no new input (contrast to the d instruction)
- processing of pattern space continues from top of script

# Multiline Pattern Spaces – Example

```
$ sed  '/Adams/{
>        N
>        s/.-[0-9]*/censored/g
>        }'  phone.list
Smith, Terry                      7-7989
Adams, Fran                       censored
StClair, Pat                      censored
Brown, Robin                      1-3745
Stair, Chris                      5-5972
Benson, Sam                       4-5587
Harris, Ford                      6-6221
Phiri, Ray                        3-3333
Llewellyn, Nia                    7-8823
$ _
```

# The Hold Space

This is a set-aside or copy buffer
Hold space cannot be directly changed (edited)
It is a temporary storage area

There are three instructions available

- h or H copy or append contents of pattern to hold space (HOLD)
- g or G copy or append contents of hold to pattern space (GET)
- x swap pattern and hold space (EXCHANGE)

An example

```
$ print "1\n2" | sed '/1/{
>                     h       hold line matching 1
>                     d       delete pattern space
>                     }
>                   /2/G'    2 line + hold space
2                               print pattern space
1
$ _
```




AU232A38

# The tar Utility

This is an archive/backup command

Historically used tape but now any device

- default to /dev/rmt0

Syntax: `tar options pathname(s)`

# tar Options

Options are of two types

- required
- optional

Should be specified using a leading hyphen

Required options are one of

- c - create an archive
- x - extract file(s) from archive
- t - list (tell) what is in archive

Other (optional) options are

- f - used to specify other than default device
- v - verbose (usually with t or x)
- m - restore/keep modification times

# tar Pathnames

*tar* takes a pathname as one of its parameters

Full pathnames mean that restores (extracts) will be to original directory

Relative pathnames mean that restores may be to any part of filesystem

*tar* may be used to do recursive copies of data from one directory to another

```
$ cd fromdir; tar cf - . | (cd todir;\
>tar xf -)
```

# Working in Absentia

You can submit jobs for execution later

AIX provides two useful utilities

- at
- crontab

Access to these facilities is controlled by the system administrator

# The at command

*at* submits a set of commands (a job) for later execution

Syntax: `at [-r|-l] time`

Commands are read from stdin

`time` can be specified as absolute or relative

- the time may include a date

Options include

`-l` list your at jobs

`-r` remove your *at* job(s)

*at* uses mail to send the stdin and stderr output (unless redirected)

System administrator determines who may use *at*

# at Usage and Examples

Here are some examples (commands excluded)

at 2100
at 10pm
at 4am
at 9am tomorrow
at 10:30 Jul 3
at now + 2 hours
at now + 2 days
at now + 1 year

# The crontab Command

This command is like *at* but for regular "jobs"

Syntax: `crontab [-e | -l | -r] [job-file]`

The commands executed are in job-file (or from stdin)

The options allow you to edit, list or remove your crontab file

System administrator determines who may use *cron*

*cron* will mail the output of the command to crontab owner

# crontab File Format

*cron* needs crontab files in a particular format

Each line has time(s)/date(s) and the command to run

Format of each line is a set of fields

- minute (0-59)
- hour (0-23)
- day (1-31)
- month (1-12)
- day of week (0-6, 0 = Sunday)

Each of the first five fields may be

- a number
- a comma separated number list (1,3,4,13)
- a range (4-9)
- an asterisk (*)

Sixth field contains the command(s) executed (a % means a newline)

# crontab Examples

Here are some possible crontab file entries/lines

```
# Run command at 0900 and 1200 Mon-Fri
15 9,12 * * 1-5 /home/sa/games_off
# Do some backups at 0200 Tue-Sat
0 2 * * 2,3,4,5,6 /home/sa/backup daily

# What does this one do?
13 5 * * 0 find $HOME -name ,\* -exec rm -f {} \;
```

# Summary

- Use of *sed* to automate repetitive editing tasks
- Archiving using *tar*
- Batching commands for later execution:
  - One off using *at*
  - Regular or repeated using *crontab*

# Unit 11. The AWK Program

# Objectives

This unit will show you how to use the awk utility by looking at:

- Regular expressions in awk
- Basic awk programming
- BEGIN and END processing
- Flow control – if, while, and for
- Leaving loops – continue, next and exit
- Awk arrays
- Better printing
- Awk functions

# What Is Awk?

- **Awk** is a programming language used to manipulate text
- **Awk** sees data as words (**fields**) in a line (**record**)
- An **awk** command consists of a ***pattern*** and an **action** comprising one or more statements

```
awk '/pattern/ { action }' file ...
```

- **Awk** tests every **record** in the specified ***file(s)*** for a ***pattern*** match. If a match is found, the specified **action** is performed
- **Awk** can act as a filter in a pipeline or take input from the keyboard (standard input) if no ***file(s)*** are specified

# Sample Data – awk

```
Lastname,<SPC>Firstname<TAB>nnn-mmmm

$ cat phone.list

Terrell, Terry                      617-7989
Franklin, Francis                 704-3876
Patterson, Pat                      614-6122
Robinson, Robin                     411-3745
Christopher, Chris                  305-5981
Martin, Marty                       814-5587
Llewellyn, Lynn                     316-6221
Jansen, Jan                         903-3333
Llewellyn, Lee                      817-8823
$ _
```

The same file as in the RE and sed units

# awk Regular Expressions

- Like sed, regular expressions are "**/**" delimited – "**/x/**"
- All of the previous regular expression metacharacters can be used with **awk**

Awk has the following extensions

| | |
|---|---|
| `/x+/` | for one or more occurrences of x |
| `/x?/` | zero or one occurrence of x |
| `/x\|y/` | matches either "x" or "y" |
| `(string)` | groups a string – for use with **+** or **?** |

Example:

```
/t[i|o]?n[iey]+/
```

matches: tiny, tony, toni, toney, tone (and others...)

# awk Command Syntax

- Basic syntax

```
pattern  { actions }
pattern
         { actions }
```

- Multiple statements in an action
  - use a line break or a semi-colon

```
$ awk '/Ll/ { print $1 ; print $3 }'\
> phone.list
```

- Comments start with a # until the end of a line

```
$ awk '/Ll/ { print $1 # prints field 1
>       print $3 }' phone.list
```

# The print Statement

One useful **action** is to **print** the data!

```
awk '/pattern/ { print }' ifile > ofile
```

- awk tests each **record** of the input for the specified *pattern*
- When a match is found the **print** statement sends the entire **record** to standard output

# awk Fields and Records

- Referencing fields in a record

  $0 = the entire record

  $1 = the first field in the record

  $2 = the second field in the record

  ...

- To print the first two fields in records beginning with "Ll"

```
$ awk '/^Ll/ {print "Name:", $2, $1 }' \
> phone.list
Name: Lynn Llewellyn,
Name: Lee Llewellyn,
$ _
```

# print Examples

- Special character sequences are available for use in print strings or regular expressions

  **\n** newline
  **\t** tab
  **\r** carriage return

```
$ awk '/^Ll/ { print "Name:\t", $1
>       print "Number:\t", $3, "\n" }' phone.list
Name:    Llewellyn,
Number:  316-6221

Name:    Llewellyn,
Number:  817-8823

$ _
```

# Comparison Operators and Examples

To compare regular expressions or strings with values:

```
==  equal to                !=   not equal to
<   less than               <=   less than or equal to
>   greater than            >=   greater than or equal to
~   matched by RE           !~   not matched by RE
||  logical "or"            &&   logical "and"
```

Examples

| | |
|---|---|
| `$1 ~ /x/` | field one matches regular expression x |
| `$1 !~ "No"` | field one doesn't match string "No" |

You can use comparison operators in the ***pattern*** to select records

```
$ awk '$1 == "Terrell," { print $2, "Smythe" }' phone.list
Terry Smythe
$ _
```

# Arithmetic Operators

You can use the following operators to perform arithmetic:

| | | |
|---|---|---|
| `+` | | addition |
| `-` | | subtraction |
| `*` | | multiplication |
| `/` | | division |
| `%` | | remainder |
| `^` | | exponential (x^y, raise x to the power y) |
| `++x` | `x++` | pre and post increment |
| `--x` | `x--` | pre and post decrement |
| `=` | | assignment (x = 4) |

```
x op= y            x = x op y
                   for:  +=, -=, *=, /=, %=
```

Example

```
count = count + 2
num *= 8
```

# User Variables and Expressions

You can define your own variables:

- Names must:
  - start with a letter or underscore
  - be followed by letters, underscores or digits
- Awk does not require variables to be defined before use

Variables are initialized as empty (numerically zero)

The empty string is null ("")

Reference by name only

# BEGIN and END Processing

You have seen the **pattern** and **action** awk syntax
You can also have actions at the beginning and end of input
You use the special patterns BEGIN and END

```
awk 'BEGIN { begin_action }
     pattern { action }
     pattern { action }
     END { end_action }'   file...
```

Where

| | |
|---|---|
| *BEGIN* | means execute the *begin_action* before any input read |
| *END* | means execute *end_action* once all input has been read |

# BEGIN without END Example

You can use **BEGIN** to print a header to the output...

```
$ awk  'BEGIN { print "Words"}
>              { wcount = wcount + NF
>                print wcount }'  phone.list
Words
    3
    6
    9
    ...
    24
    27
$ _
```

- Here we have a BEGIN with no END
- The statements within the second set of braces were performed on every line of "phone.list" as no ***pattern*** was specified



# END without BEGIN Example

You can use **END** to print a trailer after the output

```
$ awk  '{ wcount = wcount + NF }
> END { print "Words: ", wcount }' phone.list
Words: 27
$ _
```

- The statement within the first set of braces refers to the main **action**
- The main **action** is performed on every line of the file "phone.list", so the final value of wcount holds the total number of fields (or words) in the file
- At the end of the input **END** actions are processed
- This prints the heading "Words:" with the total word count

# Built-In Variables

Awk provides a number of useful built-in variables:

| | |
|---|---|
| **FILENAME** | the name of the current **file** |
| **NF** | total number of **fields** in the current record |
| **NR** | number of **records** encountered |
| **FS** | **input field separator** (the default is space or tab) |
| **RS** | **input record separator** (default is newline) |
| **OFS** | **output field separator** (default is space) |
| **ORS** | **output record separator** (default is newline) |

# Built-In Variables Examples - 1

```
$ cat employee.list
Name, company, city, phone
Pete Davis, IBM, Augusta, 770-835-3788
Bill Moran, IBM, Gaithersburg, 301-240-8068
Tommy Todd, IBM, Atlanta, 770-835-3523
$ _
```

```
$ awk 'BEGIN { FS = "," ; OFS = ":" }
>       { print $1, $4 }' employee.list
Name: phone
Pete Davis:  770-835-3788
Bill Moran:  301-240-8068
Tommy Todd:  770-835-3523
$ _
```

# Built-In Variable Examples - 2

```
$ cat authors
R.S. Davis                  FIELD 1
Augusta, GA 30809           FIELD 2
770-835-3788                FIELD 3
                            RECORD SEPARATOR
F.W. Moran
Gaithersburg, MD 20879
301-240-8068

C.T. Todd
Atlanta, GA 30339
770-835-3523

$ awk 'BEGIN { FS="\n" ; RS="\n\n" ; OFS="\n" ; ORS="\n\n"}
> { print $1, $3
> } ' authors
```




AU232B34

# if - else if - else Statement

```
awk '{
     if (first logical test) {
          action if test true
     }
     else if (second logical test)  {
          action if first test false and
          second test true
     }
     else  {
          action if both tests false
     }
}' file
```

# The while Loop

```
awk   ' {
      while   (condition)   {
                   action
      }
      } ' file
```

Example

```
awk   ' {
      i = 1
      while    (i <= 4)    {
                   print   $i
                   ++i
      }
      } ' file
```

# The for Loop

```
awk   '{
        for (initialise; test; increment) {
                action
        }
       }' file
```

**Examples...**

- **to read and print each field of the current input line**

```
for (i=1; i<=NF; i++)
          print $i
```

- **to print from the last field to the first of the current line**

```
for (i=NF; i>=1; i--)
          print $i
```

# The continue and next Statements

The **continue** statement stops the current innermost loop iteration and starts the next one:

```
awk  '{
       y = 42
       for (x=1; x<=NF; x++) {
              if (y!=$x)
              {
                   continue
              }
              print x, $x
       }
     }'  file
```

The **next** statement causes the next **record** to be read in, and the program to start from the first ***pattern* { action }** block again:

```
awk 'BEGIN { action }
     pattern {
               action
               action
               next
               action
               }
     END { action }' file
```

# The exit Statement

The **exit** statement jumps to any ***END*** processing – or out of the program if already in the ***END*** section. An exit code can be passed back to the Shell:

```
$ awk  '{
>          y = 42
>          for (x=1; x<=NF; ++x) {
>               if (y==$x) {
>                    print x, $x
>                    exit
>               }
>          }
>       }
>       END  { exit 3 }' file
$ print $?
3
$ _
```

AU232B44

# Arrays

- Awk allows array variables
- An array is a variable with an index
- An index is an expression in brackets
  - for example, array[ 10 ]

- Awk arrays are "associative"
  - index can be a string or number
  - no implicit order
  - to access all elements, use the **in** operator

```
for ( var in array_name )
```

Be aware that all array indices are internally strings

# printf for Formatted Printing

- One use of awk is as a report generator
- Better printing formats required
  - use **printf**
- **printf syntax:** `printf ( fmt [, args] )`

- Parentheses are optional
- *fmt* is usually a string constant with format specifications
- Specifiers are like the C language printf
- Format specification: %<char>

| | |
|---|---|
| %s | string |
| %d | decimal integer |
| %f,%e | floating point (fixed or exponent notation) |
| %o | unsigned octal |
| %% | literal percent |

# printf Formats

- Format specification strings can use modifiers

```
%-width.precision
```

  - If width used, contents are right justified
  - use - (minus/hyphen) after % to <u>left</u> justify
  - precision controls
    1. number of digits to right of decimal point for numeric values
    2. maximum number of characters to print for string values

- To print Hello within #'s right justified in 10 character field

```
printf ("#%10s#\n", "Hello")
```

- To print a number left justified with minimum 3 characters

```
printf ("%.3d\n", $1)
```

AU232B52

# Functions in Awk

- There are four types of functions
- Three types are built-in to awk
  - general
  - arithmetic
  - string
- The fourth type is a user defined function

- General functions include
  - close
  - system
  - getline

# Built-In Arithmetic Functions

Functions available include

| | |
|---|---|
| atan2(y,x) | arctangent of y/x in range $-\pi$ to $+\pi$ |
| cos(x) | cosine of x (x in radians) |
| sin(x) | sine of x |
| exp(x) | *e* to the power x |
| log(x) | natural log of x |
| sqrt(x) | square root of x |
| int(x) | truncated value of x |
| rand() | pseudo-random number r, $0 \leq r \leq 1$ |

# Built-In String Functions

Functions available include

| | |
|---|---|
| length(s) | length of string s or of $0 if s not supplied |
| index(s,t) | position of substring t in s or zero if not present |
| match(s,r) | position in s of where RE r begins or zero |
| sub(r,s,t), gsub(r,s,t) | substitutes for r in t, returns 1 for OK uses $0 if t not supplied (gsub does all matches) |
| split(s,a,sep) | parses s into array a elements using field separator sep (use RS if not supplied) |

Set by match()

| | |
|---|---|
| RSTART | start of the match (same as the return value) |
| RLENGTH | length of the matching sub-string |

# Summary

- Regular expressions in *awk*
- Basic *awk* programming
- BEGIN and END processing
- Flow control – if, while, and for
- Leaving loops – continue, next and exit
- Awk arrays
- Better printing
- Awk functions

# Unit 12. Putting It All Together

# Objectives

In this unit we will see:

- Shell script uses in AIX
- Program headers
- Program structure
- Selected syntax examples

# Korn Shell Scripts in AIX 4.3

/usr/sbin

| | | | |
|---|---|---|---|
| automount | bosboot | cfgmir | cfgvg |
| chC2admin | chlv | chlvcopy | chpv |
| chvg | chwebconfig | clvm_cfg | cplv |
| dhcpaction | dhcpaction8 | dhcpremove | dhcpremove8 |
| dtappintegrate | exportvg | extendlv | extendvg |
| fbcheck | importvg | index_config.sh | index_unconfig.sh |
| IsC2admin | lsjfs | migratepv | mirrorvg |
| mkC2admin | mkinsttape | mklv | mklvcopy |
| mktcpip | mkvg | piofontin | piomisc_base |
| rc.bootx | redefinevg | reducevg | reorgvg |
| rmC2admin | rmlv | rmlvcopy | shutdown |
| slipcall | snap | splitlvcopy | synclvodm |
| syncvg | tapechk | unmirrorvg | updatelv |
| updatevg | varyoffvg | which_fileset | |

/usr/bin

| | | | |
|---|---|---|---|
| bf | bfrpt | chdoclang | chlang |
| chtz | defaultbrowser | ibm3812 | mkpmhlv |
| mksysb | mkszfile | ndx | oslevel |
| pmd | restvg | smit | spellin |
| subj | vgrind | | |

/etc

| | | | |
|---|---|---|---|
| rc | rc.C2 | rc.bsdnet | rc.dacinet |
| rc.dt | rc.net | rc.net.serial | rc.powerfail |
| slip.logout | | | |

# Shell Script Uses in AIX 4.3

Shell Scripts also make up part of the AIX operating system:

Start-up and shutdown...

- `rc.*` multi-user start-up programs
- `bosboot` configures and creates a device boot image
- `mktcpip` sets required values for starting TCP/IP
- `shutdown` used to shutdown the system before power-off, or to enter maintenance mode

Documentation...

- `snap` documentation for your system

# Program Headers

```
#!/bin/ksh
#@(#)54      1.45 src/tcpip/usr/sbin/mktcpip/mktcpip, tcpip, tcpip43D, 9808A_43D 2/20/98
17:59:51
#
#COMPONENT_NAME: (TCPIP)
#
#FUNCTIONS: mktcpip.sh
#
#ORIGINS: 27
#
                      COPYRIGHTS HAVE BEEN DELETED TO SAVE SPACE
#
##[End of PROLOG]

#FILE NAME: mktcpip
#
#FILE DESCRIPTION: High-level shell command for performing minimal
# configuration required to get a maching up and running TCP/IP.
#
#Basic functions performed are:
# 1) the hostname is set both in the config database and in running machine
# 2) the IP address of the interface is set in the config database
# 3) /etc/hosts entries made for hostname and IP address
# 4) the IP address of teh nameserver and domain name are set
# 5) the subnet mask is set
# 6) destination and gateway routes are set
# 7) TCP/IP deamons started
#    or
# 8) Retrieve the above information for SMIT display
# 9) the cable type (bnc, dix or tp) is set in database
#
# See Usage message for explanation of parms
#
#RETURN VALUE DESCRIPTION
#        0        Successful
#        non-zero Unsuccessful
#
#
#EXTERNAL PROCEDURES CALLED: chdev, hostname, hostsent, lsdev
#                              mkdev, netstat, namerslv, /etc/rc.tcpi, route
```

# Program Headers (Cont.)

```
#!/bin/ksh
#/usr/sbin/mktcpip
...
PATH=/bin:/usr/bin:/usr/sbin:/etc:/usr/ucb export PATH

NAME=$0

#Parse command flags arguments
set -- `getopt h:a:i:n:d:m:g:t:r:sc:D:S: $*`
if [ $? != 0 ]; then     #test for syntax error
  usage                  #issue msg and don't return
fi

if [ $# -lt 3 ]; then    #test for too few parms

HOSTNAME= IPADDRESS= INTERFACE= NAMESERVER= DOMAIN= SUBNETMASK=
DESTINATION= GATEWAY= STARTTCP= SHOW= TYPE= DESTADDR= SUBCHANNEL=
RING=

while [ "$1" != "--" ]
do
   case $1 in
      -h)unset HOSTNAME
         HOSTNAME=$2 shift 2;;
...
```

# Program Structure

```
/usr/sbin/snap
#----------------------------MAIN-----------------------------
trap intr_action 2
# Save off current umask and set it to 077.
UMASKSAVE=`umask`
umask 077

set -- `getopt AaDd:flgGklcnNo:prv:sStXib $*`
if [ "$?" != 0 ]
then
   usage
   exit 1
fi
userid=`id -ru`
if [ "$userid" != 0 ]
then
   echo "Must be root user [0] to use this utility"
   exit 2
fi

while [ "$1" != -- ]
do
   case $1 in
      -A)    doasync=y        #Gather async (tty) information
         action=y
         shift;;
      -a)    doall=y          #Gather all information
         dopred=y
         dosec=y
         action=y
         shift;;
      -d)    destdir=$2       #Directory to put information
         valid_dir $destdir
         shift;shift;;
...
```

# Selected Syntax Examples - 1

Rather than wade through very long programs, here we have some selected interesting bits of syntax

**rc.net:** using exec & re-direction...

```
#  Close file descriptor 1 and 2 because the parent may be
# waiting for the file desc. 1 and 2 to be closed.  The reason
# is that this shell script may spawn a child which inherits
# all the file descriptors from the parent and the child
# process may still be running after this process is
# terminated.  The file desc. 1 and 2 are not closed and leave
# the parent hanging waiting for those desc. to be finished.

LOGFILE=/tmp/rc.net.out     # LOGFILE is where all stdout goes.
>$LOGFILE                   # truncated LOGFILE.

exec 1<&-                   # close descriptor 1
exec 2<&-                   # close descriptor 2

exec 1</dev/null            # open descriptor 1
exec 2</dev/null            # open descriptor 2
```

# Selected Syntax Examples - 2

```
#!/bin/ksh
#/usr/sbin/snap
...
  TMPDIR=${TMPDIR:-$HOME/tmp}
  [[ ! -d $TMPDIR ]] && TMPDIR=/tmp
  TMPDIR=$TMPDIR/${0##/}.$$

  mkdir $TMPDIR || {
    print -u2 "${0##*/}: Could not create temporary files"
    exit 1
  }
  trap "/bin/rm -rf $TMPDIR 2>/dev/null" EXIT INT TERM QUIT HUP

  tdumpf=$TMPDIR/tmpfile.$$
  ...
```

# Selected Syntax Examples - 3

/usr/sbin/snap

```
...
#
#Now proceed to call the associated functions for real
#This is pass 2 on state functions
passno-2
for i in $state
do
   state_func${i}
done

#Set the umask back to the original value
umask $UMASKSAVE
...
```

shutdown **sed** & **awk** example...

```
# NAME: tabmnt
# FUNCTION: collect the mount information and force every field
# to be separated by a tab, so that awk can look at the
# different fields.
tabmnt()
{

mount 2>/dev/null | awk '{ line[i] = "-"$0; i++; }
                    END { while ( i >= 4 ) {
                          i--; print line[i]; }
                         }' - >/tmp/mount.a

tab /tmp/mount.a
# remove extra tabs and blanks

   sed "/ /s//       /g" /tmp/mount.a \
   | sed "/                    /s//       /g" \
   | sed "/                    /s//       /g" \
   | sed "/         /s//         /g" >/tmp/mount.t
}

rm -f /tmp/mount.a 2>/dev/null
```

# Selected Syntax Examples - 4

```
#!/usr/bin/ksh
# /usr/sbin/cfgmir
...
  #keep getting parent device until parent device is a bus
  #device or sio device
  print $PARENT_MON | egrep "bus|sio" > /dev/null 2>&1
  done ="$?"
...
  #wait (with timeout) the end of portmir
  for i in 1 2 3 4 5 6
  do
    if ps -ef | grep portmir | grep -v grep >/dev/null
    then
      sleep 1
    else
      break
    fi
  done
...
```

# Summary

- Shell Script uses in AIX
- Program headers
- Program structure
- Selected syntax examples

# Unit 13. Good Practices and Review

# Objectives

To write any serious script we need to:

- plan the activity
- produce "good code"

In this unit:

- Planning and design
- Documentation
- Debugging
- Performance issues
- Guidelines for scripting
- Course summary

# Planning and Design

As well as your favorite design methodology (Flow Charts, Data-Flow, SSADM, etc.) consider:

- functionality – clearly defined specification
- modular design – use of functions, separate programs
- environment – variables, directories
- file naming convention – for temporary files, results
- testing – individual units, integration tests, boundary conditions
- debugging code – do not forget the next maintainer

# Use of Comments

A good programmer uses comments in a program to:

- Explain the purpose and function of the code at key points
- Describe the use of variables
- Explain complicated syntax
- Give yourself the credit (or the blame) for your work
- Mark corrections or additions

Remember to update the comments with the code

# Commenting Out

Lines can be commented out using the # comment character:

```
# command arg1 arg2
```

- no Shell interpretation is performed to the right of #
- legal anywhere, except as the only statement in a flow-control construction (if, while, until)

The "null" command can be used where commenting out would not work:

```
:  command arg1 arg2
```

- arguments are ignored, but processed as usual
- always returns 0 (true)

# Script Layout

Some things must be done in a certain order
other things can be arranged for "good code":

- Shell control line (first in script) `#!/usr/bin/ksh`
- Header comments
- Validation of options
- Testing of arguments
- Initialization of variables
- Function definitions
- Main code

# Debugging Code

Korn Shell options can help with syntax checking:

- to check the syntax of a Shell Script without running it

  `set -o noexec` or `set -n`

- for the Shell to print its input as it reads it

  `set -o verbose` or `set -v`

- an execution trace displays each command <u>before</u> it is run and <u>after</u> command line processing

  `set -o xtrace` or `set -x`

- for functions, use

  `typeset -ft function ...`

# DEBUG Traps

After each simple command the Korn Shell issues the fake signals

- **DEBUG**
- ERR
- EXIT

The order is DEBUG, ERR, then any other traps, and lastly EXIT

To display the environment after each command set this trap

```
trap  "set"  DEBUG
```

When a command has a non-zero exit status, the Korn Shell sends the **ERR** signal

For example, to see what signals are causing error exits set this trap

```
trap  "kill -l $?"  ERR
```

# Maintaining Code

Documentation: design and comments

Clarity

- Code
- Documentation

Modularity

- Main script
- Use "good" functions or separate programs

# Good Functions

To write functions that are reliable and easy to maintain:

- avoid altering global variables inside a function
- define and export functions only when necessary
- do not change the working directory inside a function (why?)
- tidy up local temporary files

# Performance Issues for Shell Scripts

If performance is an issue

- Do not guess
- Measure!

Performance of a script means two areas:

- that of the Korn Shell
- that of the script

Remember that you should work in this order

- Get the functionality working
- Make it robust
- If you have to, make it more efficient/faster

# Timing Commands

To report the elapsed, user and system time for a command or pipeline, use **time** in the KornShell:

- a Korn Shell reserved word (not a command)
- **time** output is to standard error
- input or output redirection applies to the command(s) under test only
- return value is that of the command(s) under test

```
$ time find / -name 'unix*' -print|sort
/unix                      find output
/usr/lib/unixtomh
real   0m25.51s            wall clock time
user   0m1.56s
sys    0m11.01s
$ _
```

# Times for Shells

The **times** command displays how much time your current Shell and all its Sub-Shells have consumed:

```
$ times
0m0.99s 0m15.37s
0m8.61s 0m33.21s
```

- user and system timings given in hundredths of a second
- first line for the current Shell
- second line for the Sub-Shells

# Korn Shell Performance

To increase the startup speed of a new Shell:

- keep your history file **(.sh_history)** small
- minimize the size of any **$ENV** file
- use *autoload* with your functions
- use *FPATH* with your functions
- `set -o nolog` to prevent function definitions being logged in your history
- use "tracked aliases"
- try to use an **alias** in place of a simple function
- set *MAILCHECK* greater than the 600 second default

# Korn Shell Script Performance

Tips for faster performance Shell Scripts:

- Shell built-in commands run faster than AIX ones
- Avoid command substitution where you can use `${ }` parameter expansions, *let* or pattern matching
- Note `$(< file)` is faster than `$(cat file)`
- Use multiple arguments rather than separate commands – e.g. `typeset -i a=3 b=4`
- Use `set -f` **or** `set -o noglob` if not using pathname metacharacters
- Use `{ }` grouping that is faster than `( )`
- Apply I/O re-directions to the whole of a loop syntax
- Set the *integer* attribute for suitable variables and don't use **$** for them with arithmetic expressions



# Good Rules To Follow

1. Documentation
2. Make Backups
3. Try three times
4. Don't overlook the obvious
5. Try it, it might work
6. Never say never, always avoid always
7. There's usually another way to do it

## Course Summary

Basic concepts

Shell variables and parameters

Exit status, return codes and traps

Progamming constructs – control flow

Shell commands and features

Arithmetic in Shell

Shell types and functions

# Course Summary (Cont.)

- More Shell variables
- Regular expressions and text selection
- Personal productivity – *sed*, *crontab/at*, *tar*
- Using *awk*
- Shell scripts in practice
- Summary – good practice, debugging, performance

# Summary

- Planning and design
- Documentation
- Debugging
- Performance issues
- Guidelines for scripting
- Course summary