

AFRL MSRC Bourne Shell (sh)  
Programming Tutorial

## Topic List

|   |    |
|---|----|
| Introduction . . . . .                                | 1  |
| Basic shell script . . . . .                          | 2  |
| Variables . . . . .                                   | 3  |
| Useful variable values . . . . .                      | 6  |
| Constructing strings . . . . .                        | 7  |
| True or false . . . . .                               | 8  |
| test command . . . . .                                | 8  |
| Common test operators . . . . .                       | 9  |
| : command . . . . .                                   | 10 |
| Making choices (if and case) . . . . .                | 11 |
| Positional parameters . . . . .                       | 17 |
| Looping: for, while, until . . . . .                  | 19 |
| Doing math . . . . .                                  | 23 |
| eval commmand . . . . .                               | 24 |
| /bin/true, /bin/false, continue, break . . . . .      | 25 |
| find command . . . . .                                | 26 |
| Useful find options . . . . .                         | 26 |
| read command . . . . .                                | 27 |
| In-line input redirection or here documents . . . . . | 29 |
| Functions . . . . .                                   | 31 |
| . command . . . . .                                   | 34 |
| exit command . . . . .                                | 34 |
| Stream manipulation . . . . .                         | 35 |

[This is intended to be a tutorial on scripting in the Bourne sh shell. The interested reader is referred to Kochan and Wood, *Unix Shell Programming*, and similar texts.]

## Bourne Shell (sh) Programming

Note: ‘ ’ denotes a space (‘ ’) character.

Bourne shell (sh) – the scripting language of Unix system administrators

Also, the root shell language of Korn shell (ksh), and Bourne-again shell (bash)

As for C shell (csh) and its derivatives (tcsh), see the document <http://www.faqs.org/faqs/unix-faq/shell/csh-whynot/>.

Many scripts for system administration tasks are written in Bourne sh due to the limited resources available within the single-user maintenance mode.

A comparison of Bourne-derived shells and C-shell derivatives can be found at <http://www.faqs.org/faqs/unix-faq/shell/csh-whynot/>.

The following are two examples of a simple shell script. As the reader can see, this is a series of commands to change to the user's \$HOME directory, create a subdirectory, and print out the line from the `env` command containing the string `PATH`; the reader could execute these same series of commands from the command line.

### Basic shell script

---

```
#!/bin/sh
# Basic shell script
cd $HOME
mkdir foo
env | grep PATH
```

---

or

---

```
#!/bin/sh
# Basic shell script
cd $HOME ; mkdir foo
env | \
    grep PATH
```

---

Note: commands can be separated by a newline character or ‘;’. ‘\’ is used for line continuations.

The first line in the file “#!/bin/sh” tells the shell what interpreter to use with the script as input. For Bourne sh scripts only, this line is optional as the behavior is to use the default Bourne sh in the user's path as the interpreter. Due to space considerations, the “#!/bin/sh” line will be omitted in the examples as they are code snippets. The second example demonstrates the use of “;” to combine commands on the same line and “\” to split command lines on multiple lines.

Any text following a “#” character is treated as a comment.

These scripts must have the execute-bit enabled (`chmod u+x filename`) to execute these scripts as commands.

The utilities test, expr, and find will be used in the examples. These utilities have individual man pages.

The following are a few utility commands that will be used in subsequent examples.

test – logical operations, file information  
expr – integer arithmetic operations  
find – locates files with specific attributes

As with other programming languages, variables provide a means of storing and recalling data via a naming scheme.

## Variables

*name* must begin with a letter or ‘\_’ character, followed by zero or more letters, numbers, and ‘\_’ characters.

Assignment: *name=value* (no spaces around ‘=’)

Environment variable: `export name-1 [ ... name-N ]`

Value: *\$name*

Replaces with value: *\$name*, “*\$name*”

Executes in shell and replaces with output: ``$name``

No change: `\$name`, `'$name'`

In assigning a variable, there is no whitespace around the “=” operator. export changes an ordinary shell variable into an environment variable, which is passed to subshells or child shells.

The value of a shell variable is referenced by prepending a “\$” to the name. Note that using double quotes (") or grave quotes (`) expands to the variable’s value before its use whereas escaping the “\$” or using single quotes (‘) does not.

The example illustrates the various methods of assigning and using shell variables, with the output following. The `pwd` command outputs the full path of the current directory. The `echo` command prints its arguments to standard output. The `env` command prints out the names and values of the environment variables in the current shell process. The `grep` command prints out occurrences of its first argument in its input (standard input in the example).

### Example: comparing shell and environment variables

---

```
pwd
```

```
foo="Value of foo"
```

```
AKIRA="pwd"
```

```
export AKIRA
```

```
echo $foo "$AKIRA"
```

```
echo \$foo `$AKIRA`
```

```
echo ` $AKIRA `
```

```
env | grep AKIRA
```

```
env | grep foo
```

---

Result:

---

```
/home/user
```

```
Value of foo pwd
```

```
$foo $AKIRA
```

```
/home/user
```

```
AKIRA=pwd
```

```
<blank line>
```

The script: 1) outputs the full path of the current directory, 2) assigns values to the shell variables **foo** and **AKIRA**, 3) makes **AKIRA** an environment variable which is available to child processes, 4) outputs the various forms of the two shell variables, and 5) outputs matches of the two shell variables in the environment variable output.

The first line of the output is the full path of the current directory.

The second line of the output is the concatenation of the values of the two shell variables. This shows that both the unquoted form and quoting with double quotes ("") expands to the value of the shell variable.

The third line of output are the variable names preceded by a '\$' character. Escaping (preceding by '\') the '\$' or enclosing in single quotes (``) prevents expansion to the value of the shell variable. This form is useful if '\$' does not precede a valid shell variable name, *e.g.*, currency expressions. Delayed expansion is also useful when the shell variable has no value in the current shell process (*e.g.*, writing out a shell script, executing a command on a remote system).

The fourth line of output is the full path of the current directory. The grave quotes (``) execute the value of the shell variable as a command in a subshell. Here, **\$AKIRA** expands to **pwd**, and the output of the command is stored as a string.

The last two lines of output (the last line is blank) show that **AKIRA** was defined as an environment variable whereas **foo** was not.

The following is a list of common shell variables. Some of these will be used in later examples.

### Useful variable values

|                                |  |
|--------------------------------|--|
| <code>\$PATH</code>            | colon-separated list of directories in which to search for commands                                    |
| <code>\$LD_LIBRARY_PATH</code> | colon-separated list of directories in which to search for shared objects                              |
| <code>\$MANPATH</code>         | colon-separated list of directories in which to search for man pages                                   |
| <code>\$HOME</code>            | user's home directory  |
| <code>\$USER</code>            | login name   |
| <code>\$PWD</code>             | full path of current directory   |
| <code>\$?</code>               | exit status value of last command executed   |
| <code>\$\$</code>              | process id (PID) of the current process  |
| <code>\$0</code>               | name of called program   |
| <code>\$1-\$9</code>           | up to the first nine positional parameters   |
| <code>\$#</code>               | count of positional parameters<br>(can be larger than nine)  |
| <code>\$*</code>               | string of all the positional parameters  |
| <code>\$@</code>               | same as <code>\$*</code> except " <code>\$@</code> " leads to all parameters being individually quoted |

Of note for script writing is the `$@` variable. When writing scripts that pass command-line arguments onto other commands, it is useful to preserve the original arguments, which may have embedded characters that are special to the shell. Using "`$@"`", a script will pass the command-line arguments as being individually quoted.

## Constructing strings

Assume that the value **\$foo** is the string “JIRO,” the value **\$bar** is the string “mohmoh-rehnjah,” and the value **\$null** is the empty string (“”) in the following examples.

Expression: `$foo5`

Equivalent: “” [undefined variable]

Expression: `${foo}5`

Equivalent: “JIRO5”

Expression: `x$bar`

Equivalent: “xmohmoh-renjah”

Expression: `'x$null'`

Equivalent: “x\$null” [no expansion]

Expression: `$bar 'has explosive jewelry.'`  
“\$bar has explosive jewelry.”

Equivalent: “mohmoh-renjah has explosive jewelry.”

Expression: ``echo ${foo} “5$null”``

Equivalent: “JIRO5” [result of echo evaluation]

These examples demonstrate constructions of strings which combine literal strings and shell variable values. Except in the case of characters that have special meaning in the shell (*e.g.*, spaces, dollar signs), quoting in strings is not necessary. However, quoting does avoid errors with strings used as operands.

The first two examples demonstrate the wrong and correct methods of concatenating a shell variable value and an alphanumeric string. Since **foo5** is a valid variable name, **\$foo5** is syntactically correct even though the variable has no value. By surrounding the name with braces, the shell interpreter expands the value **\$foo**, and concatenates the string “5” to it.

In the third example, the value **\$bar** is concatenated to the string “x.” In this case, the braces are not needed.

The fourth example shows that single quotes prevent expansion of shell variable values.

The fifth example shows equivalent combinations of shell variable values and literal strings.

Although the sixth example is contrived, it shows that any of the constructed strings are valid arguments to commands

## True or false

In Bourne sh, zero (0) is true and not zero is false. This is consistent with Unix commands returning zero on success and not zero on failure. Thus, the result (not the output!) of any command, including test, can be used in decisions (if, while, until).

The constructs if, while, and until require a command return value of true (0) or false (not 0) to make decisions. Although the test command may be considered for conditional expression, as the reader can see, the return value of any command may be used in this regard.

The test command is described with its equivalent form using “[].” The reader should note that the operand of “[” is surrounded by space characters on both sides.

## test command

Although the exit status of any arbitrary command can be used to make a true/false decision, the test command outputs the result of logical and file status expressions. The test command is denoted as:

```
test expr or [ expr ]
```

where the surrounding spaces are part of the command, *e.g.*,

```
test -s "$file" is equivalent to [ -s "$file" ] ,
```

and *expr* is one or more expressions using the test operators below. Although quoting shell variables is not necessary, it avoids syntax errors when the value is the empty string.

A list of common operators for test follows. The reader should note that the logical and relational operators are not those associated with math or programming languages.

## Common test operators

|   |   |
|---|---|
| <code>-n <i>value</i></code>                            | true if <i>value</i> is not the empty string                        |
| <code>-z <i>value</i></code>                            | true if <i>value</i> is the empty string                            |
| <code>-f <i>value</i></code>                            | true if <i>value</i> is an ordinary file                            |
| <code>-d <i>value</i></code>                            | true if <i>value</i> is a directory                                 |
| <code>-s <i>value</i></code>                            | true if file <i>value</i> is not empty                              |
| <code>-r <i>value</i></code>                            | true if \$USER has read-permission on file <i>value</i>             |
| <code>-w <i>value</i></code>                            | true if \$USER has write-permission on file <i>value</i>            |
| <code>-x <i>value</i></code>                            | true if \$USER has execute-permission on file <i>value</i>          |
| <code><i>value-1</i> = <i>value-2</i></code>            | true if strings are equal   |
| <code><i>value-1</i> != <i>value-2</i></code>           | true if strings are not equal                                       |
| <code><i>value-1</i> -eq <i>value-2</i></code>          | true if integer values are equal                                    |
| <code><i>value-1</i> -ne <i>value-2</i></code>          | true if integer values are not equal                                |
| <code><i>value-1</i> -gt <i>value-2</i></code>          | true if <i>value-1</i> is greater than <i>value-2</i>               |
| <code><i>value-1</i> -lt <i>value-2</i></code>          | true if <i>value-1</i> is less than <i>value-2</i>                  |
| <code><i>value-1</i> -ge <i>value-2</i></code>          | true if <i>value-1</i> is greater than or equal to <i>value-2</i>   |
| <code><i>value-1</i> -le <i>value-2</i></code>          | true if <i>value-1</i> is less than or equal to <i>value-2</i>      |
| <code>! <i>expression</i></code>                        | true if <i>expression</i> is false                                  |
| <code><i>expression-1</i> -a <i>expression-2</i></code> | true if both <i>expression-1</i> and <i>expression-2</i> are true   |
| <code><i>expression-1</i> -o <i>expression-2</i></code> | true if at least <i>expression-1</i> or <i>expression-2</i> is true |
| <code>\( , \)</code>                                    | group expressions to increase precedence                            |

Examples:

1. test “x\$LIBPATH” = “x”
2. [`□-x /bar/RECORD.USAGE□`]
3. [`□-n “$LD_LIBRARY_PATH”□`]
4. test \$# -gt 0
5. [`□! -d ./foo□`]

Explanation:

1. test if **\$LIBPATH** is the empty string (“”)
2. test if **/bar/RECORD.USAGE** is an executable file
3. test if **\$LD\_LIBRARY\_PATH** is not the empty string (“”)
4. test if the number of command-line arguments is greater than zero (0)
5. test if **./foo** is not a directory

: command

When you need a command that does nothing, use the `:` command.

As will be seen, there are situations where a null command is useful; the `:` command serves this purpose.

## Making choices (if and case):

```
if test-command
then command-1
    [ :
      command-N ]
[ elif test-command
then command-1
    [ :
      command-N ] ]
[ else command-1
    [ :
      command-N ] ]
fi
```

---

The if construct sequentially evaluates each *test-command* until 1) one has an exit status of 0 (true), 2) the else clause is reached, or 3) fi (end of the construct) is reached. For cases 1) and 2), all commands are executed up to the next elif, else, or fi.

The reader should note that the condition for selection is the return value of a command.

```
case value in
  case-a1 [ | ... | case-aN ] )
    action-1
    [   :
      action-N ]
    ;;
[ :
  case-N1 [ | ... | case-NN ] )
    command-1
    [   :
      command-N ]
    ;; ]
[ *)
  command-1
  [   :
    command-N ]
  ;; ]
esac
```

---

Note: *value* is typically the value of a variable ( $\$name$ ) or the output of a command (``command``) where the grave (‘`’) turns the output of a command into a string. “;;” denotes the end of the actions for the group of cases.

*value* is compared to all of the *case-\** terms; if a match is found, the commands up to the terminating “;;” are executed. If no match is found, the default “\*” is matched, if it exists, and the associated commands are executed.

In the example, the condition checks if the standard input stream is connected to a terminal without generating output. Note the “;” which removes the requirement for then to be on a separate line.

### Example:

This sets the erase character to backspace only when connected to a terminal; stty will give an error when the stdin is not a terminal. tty tests stdin.

```
if tty -s ; then
    stty erase '^H'
fi
```

This snippet is useful in **.profile** file which is executed at the beginning of a login session; the clause prevents execution for non-interactive logins such as batch processing.

The command ping gives a simple test of connectivity to a remote machine. The command rsh allows command execution on remote machines if user **\$USER** has login access. However, different operating systems have differing full paths to these utilities. The following example shows one method to simplify this process for use later in a script

The output of the uname command gives the operating system name; this is assigned to the shell variable **OS**. This case construct selects the values to assign to the shell variables **PING** and **RSH** for the operating systems that it recognizes. The value **\$OS** is compared to the various case strings; the “\*” in all the cases matches any string. The last case matches all strings, and matches any case not previously matched.

By assigning values to shell variables, the commands that follow this case construct can use the values of the shell variables as opposed to the actual full pathnames. Thus, the same script can function on multiple machines minimizing maintenance errors.

Example:

Sets shell variables with full pathnames for ping and rsh

---

```
OS=`uname -s`
```

```
case $OS in
```

```
IRIX*)
```

```
    PING=/usr/etc/ping
```

```
    RSH=/usr/bsd/rsh
```

```
    ;;
```

```
OSF1*)
```

```
    PING=/usr/sbin/ping
```

```
    RSH=/usr/bin/rsh
```

```
    ;;
```

```
SunOS*)
```

```
    PING=/usr/sbin/ping
```

```
    RSH=/usr/bin/rsh
```

```
    ;;
```

```
Linux*)
```

```
    PING=/bin/ping
```

```
    RSH=/usr/bin/rsh
```

```
    ;;
```

```
*)
```

```
    echo "Unknown operating system"
```

```
    exit 1
```

```
    ;;
```

```
esac
```

As with other programming languages, case constructs can be the body of if constructs, and *vice versa*.

### Example:

A method to set the CPU environment variable for known machines

---

```
if test -z "$CPU"; then
  case `hostname` in
    aaa-[0-9]* | bbb-[0-9]*) CPU=alpha ;;
    ccc-[0-9]*) CPU=beta ;;
    ddd) CPU=gamma ;;

    *)
      echo `hostname` "does not define CPU environment \
        variable"
      echo "Contact service@host.domain about this \
        problem"
      exit 1
      ;;
  esac
  export CPU
fi
```

The snippet first checks whether the value **\$CPU** is the empty string. As the **-z** operator expects an argument, **\$CPU** is double-quoted to avoid problems when the value is the empty string.

If the value is the empty string, the output of the hostname command is compared to the cases. Here both the **"\*"**, which matches zero or more characters, and the range operator (here denoting a digit) are used in the patterns to match. If a pattern matches, **CPU** is assigned the associated value. If no pattern matches, a message is printed to standard output, and the script exits.

At the end, the variable **CPU** becomes an environment variable.

The command `ranlib` adds an index of symbols related to relocatable object files in an archive library (created by the `ar` command). SGI IRIX does not support `ranlib` as the index is created by the `ar` command during library creation/update.

Example:

When `ranlib` does not exist [or uses of the `:` command]

```
case $CPU in
  alpha | beta)
    RANLIB=: ;;
  *)
    RANLIB=ranlib ;;
esac
$RANLIB snafu.a
```

In this script, the `RANLIB` variable is assigned a value dependent on the value `$CPU`. `$RANLIB` is the command applied to `snafu.a`. Although the operation could have been done all under the specific case, this simple case can be extended to the situation where multiple archive libraries in different parts of the script are the operands.

If `$RANLIB` was the empty string, “`$RANLIB snafu.a`” would expand to “`snafu.a`” and generate a syntax error (`snafu.a` is not a valid command). Instead, for the case of SGI IRIX systems, “`$RANLIB snafu.a`” expands to “`: snafu.a`” which is a valid command that does nothing.

## Positional parameters

At a basic level, the positional parameter values `$1 ... $9`, `$*` and `$@`, and their respective count  `$#`  are the references to the command-line arguments of the script; `$0` is the name of command that invoked the script. The initial nine positional parameter values (where applicable) are referenced by the values `$1 ... $9`.

In addition, the command

```
set arg-1 ...arg-N
```

does the same for the values `arg-1 ... arg-N`. As noted previously, `$*` and `$@` refer to the entire list, and “`$@`” is useful when quoting every element in the list is necessary.

If more than nine (9) positional parameters exist, the remaining positional parameters can be moved into the `$1 ... $9` values using the shift command. With no argument, the  $k+1^{st}$  parameter is moved to the  $k^{th}$  position. With a numerical argument  $n$ , the the  $k+n^{th}$  parameter is moved to the  $k^{th}$  position.

Although both `$*` and `$@` refer to the all of the positional parameter values as space-separated lists, “`$@`” has the additional property that the individual elements are each quoted separately as opposed to quoting the list itself.

The set command allows an arbitrary space-separated list of strings to be assigned to the positional parameters. This provides an alternative method to iterating among values generated within the script. All values (`$1 ... $9`, `$*`, `$@`,  `$#` ) are modified.

To illustrate the use of the shift command, the list of positional parameter values is considered to be a space-separated list with the lowest index on the left. The command

```
shift [n]
```

will shift all the elements in the list by **n** positions (or by 1 position if **n** is absent) to the left, losing those values to the left of the first position. In addition,  `$#`  is decremented by **n** (or 1) as well.

This example demonstrates the passing of command-line arguments to another command. This snippet could be used in a wrapper script where the setup operations or logging might be incorporated in addition to executing the command. As these operations would generally execute in a subshell, the calling shell environment would not be affected.

### Example:

Passing the command-line arguments to another command, maintaining the quoting if necessary.

```
command=`basename $0` # Gets name of command
/software/$CPU/bin/$command "$@"
```

The assignment of **command** uses the basename command to get the name of the script as it was called, less any directory path information. The grave quotes (``) change the output of the command into a string for the assignment.

In a wrapper script, the value **\$0** would be the symbolic link associated with the actual script, *e.g.*, the link **snafu** created by

```
ln -s wrapper.sh snafu
```

would start the executable (`/software/$CPU/bin/snafu` when invoked. This allows one script to create a consistent operating environment for multiple related executables.

One problem with a wrapper script is that executing the script removes one level of quoting from the command-line arguments. Especially in the case of characters special to the shell (*e.g.*, \*, ?, [, ], and embedded spaces), the command-line arguments passed to the actual application may differ from what the user entered if the quote level is not maintained. To ensure that every command-line argument is passed intact (aside from expansions done before the script started), the “\$@” as the argument to the actual application command expands to the list of command-line arguments individually quoted. Note: this type of argument-form-retention is more cumbersome in csh-derived scripts.

## Looping: for, while, until

---

```
for name in value-1 ... value-N
do
    command-1
    :
    command-N
done
```

---

For each iteration of the loop, variable *name* is successively assigned *value-1* through *value-N*, and the commands in the body are executed.

---

```
while test-command
do
    command-1
    :
    command-N
done
```

---

```
until test-command
do
    command-1
    :
    command-N
done
```

---

The while and until construct loop over the commands between the do and done based on the exit status of *test-command*; they differ in behavior in that while continues looping if the exit status is 0, and until continues if the exit status is 1. Note: “;” is also a command separator.

Example:

This creates links in the system-specific directory in the default PATH which associates commands with a wrapper script.

---

```
cd /base_${CPU}
for file in `cd /software/${CPU}/bin ; ls`; do
  if [ ! -f $file ] ; then
    ln -s /software/scripts/app.sh $file
  fi
done
```

After changing to the directory for the links, the for loop iterates over the names of executable files; by changing to the directory before executing the directory listing, only the filenames are output. Note: the commands within the gravé quotes are executed in a subshell, and do not directly affect the current process.

To avoid error messages from ln, an if construct checks that the link does not already exist. The wrapper script (called **/software/scripts/app.sh**) is linked with the name of an executable binary (**\$file**).

### Example:

Changes the uppercase letters in the names of a group of files to lowercase

---

```
for file in `ls`; do
    new_file=`echo $file | tr '[A-Z]' '[a-z]`
    if test "$new_file" != "$file" ; then
        mv $file $new_file
    fi
done
```

**\$file** successively takes values of the filenames in the current directory. **new\_file** is assigned the value **\$file** with all the uppercase letters changed to their lowercase equivalent; as **tr** only works on streams, the value **\$file** is echo'ed and piped to **tr**.

The values **\$new\_file** and **\$file** are compared. If they are not equal, the file **\$file** is renamed **\$new\_file**. Both **\$file** and **\$new\_file** are quoted as a precaution against empty-string values.

This example demonstrates the while loop which terminates when its condition returns false (not 0).

### Example:

Iteratively operating on positional parameters

---

```
while test $# -gt 0 ; do
    echo $1
    shift
done
```

**\$#** is the number of positional parameters, so the loop continues as long as it is non-zero. For each loop iteration, the first parameter (**\$1**) is output, and each parameter is moved one position to the left with **\$1** being discarded. **\$#** is updated with each shift command execution.

Although this loop only prints out each parameter, the action could be more involved such as moving files or processing terms.

This example demonstrates the until loop, which terminates when its condition is true (0).

Example:

Getting a successful file transfer

---

```
STATE=-1
until test $STATE -eq 0; do
    rcp foo.dat host:/home/user
    STATE=$?

    if test $STATE -ne 0; then
        until kinit -l 10h ; do : ; done
    fi
done
```

The outer loop continues until the value **\$STATE** is 0. This loop could also be written as a while loop, which continues while the value **\$STATE** is not 0. Note: the value **\$STATE** is initialized to be false (not 0).

The body of the outer loop: 1) tries to copy a file to a remote host, 2) assigns the return value of the copy command to **STATE**, and 3) performs additional processing if the copy failed (value **\$STATE** is not 0).

The until loop in the if construct may appear counterintuitive because the loop body does nothing, and termination occurs when the command is successful. The assumption is that the remote copy command **rcp** should only fail in a recoverable way: if there are no valid credentials (loss of network connectivity is not a recoverable error). Thus, when a failure occurs, the loop executes until the credentials have been established. As the command is the condition, the body contains the no-op `:`.

## Doing math

Bourne sh does not have built-in math functions. For integer math, there is the expr command. Note: the multiplication operator is `\*`, not `*`, as the latter is special to all shells. expr has additional operators besides the usual math operators.

The expr command takes as its arguments the expression to be evaluated where the operands and operators are space-separated. Although other operators are noted in the man pages, for this discussion, only the mathematical operators `+`, `-`, `\*`, `/`, and `%` are considered.

This example tests the scalability of an MPI program. The program **foo** is assumed to be an MPI executable that can be started by the command mpirun.

## Example: scaling test

```
count=1
while test $count -le 128 ; do
    mpirun -np $count foo
    count=`expr $count \* 2`
done
```

The variable **count** stores the processor count, and begins at 1. The loop continues while the value **\$count** is less than or equal to 128.

For each loop iteration, **foo** is run on **\$count** processors, then **count** is assigned the value of twice **\$count**. As seen in previous examples, the grave quotes (```) change a command's output into a string, and do not prevent expansion of shell variables.

## eval command

When performing the same operation on a collection of variables, it is convenient to use the looping structures that work when the value of a variable is allowed to change. The eval command evaluates its argument, then executes the expanded string as a command.

Example: sets the value of the variable to its name

---

```
for match in rusage select test; do
    eval "$match"=\$match
done
```

This apparently trivial example demonstrates the utility of the eval command. The value of **\$match** is successively set to three values. On the left-hand-side of the assignment, the double-quoted string expands to the value **\$match**. On the right-hand-side of the assignment, the `\` prevents expansion of **\$match**. So, before applying eval, the argument would look like (for **\$match** being rusage):

```
"rusage"=\$match
```

which is clearly an illegal form. eval removes one level of quoting, and evaluates the result. Thus the command to evaluate would be:

```
rusage=$rusage
```

From this trivial example, other sources might be used to assign **\$match**. As this is the variable name to be assigned, a similar eval statement would assign a changing variable name to a string.

/bin/true, /bin/false, continue, break

There are cases where it is convenient to have a series of commands continue indefinitely. /bin/true and /bin/false can be used to create infinite loops with the while and until commands.

It is convenient to avoid executing all the commands in a loop if some condition is met. continue stops the current iteration of a loop and moves to the next one. break terminates the current loop.

Example:

Changes the uppercase letters in the names of a group of files to lowercase (using continue)

---

```
for file in `ls`; do
    new_file=`echo $file | tr '[A-Z]' '[a-z]`
    if "$new_file" = "$file" ; then
        continue
    fi
    mv $file $new_file
done
```

**\$file** successively takes values of the filenames in the current directory. **new\_file** is assigned the value of **\$file** with all the uppercase letters changed to their lowercase equivalent; as tr only works on streams, the value of **\$file** is echo'ed and piped to tr.

The values **\$file** and **\$new\_file** are compared. If they match, no change is necessary, and the continue statement returns control to the top of the loop. Otherwise, the filename changes to **\$new\_file**. This is a rewriting of a previous example to illustrate the use of the continue statement.

## find command

The find command is useful for locating files with specific attributes. By default, find generates a list of files that match the search criterion. The -exec option specifies a command to execute where “{” denotes the current match; the argument to -exec must be terminated by “\;”.

The reader should reference the man pages for find.

Useful find options:

|   |   |
|---|---|
| <code>-name <i>filename-specification</i></code>  | finds filenames matching the specification  |
| <code>-perm <i>permission-specification</i></code>  | finds files with matching permissions   |
| <code>-ctime <i>number-of-days</i></code><br><code>-mtime <i>number-of-days</i></code><br><code>-atime <i>number-of-days</i></code> | finds files with creation, modification, access times in <i>number-of-days</i> days |
| <code>-type <i>type-specification</i></code>  | finds files of the specified type   |
| <code>-exec <i>command</i> \;</code>  | executes <i>command</i> with “{” (the current match) as an argument.                |
| <code>! <i>option</i></code>  | negates its argument  |
| <code>\(, \)</code>   | groups options  |
| <code><i>option-1</i> -a <i>option-2</i></code>   | both specifications must be true for match  |
| <code><i>option-1</i> -o <i>option-2</i></code>   | either specification must be true for match   |

```
find . -name '*.c'    # Finds C source files
# Changes permissions to allow group/world read access find
/software/ -perm 0600 -exec chmod go+r {} \;
```

The first example locates files that end in `‘.c.’` As `‘*’` is special to the shell, it is quoted so that the wildcard is not expanded, but passed to `find` as a parameter.

The second example finds files in the directory `/software` that have only user read-write permissions. The identified files then have group and other read permissions added. In the `chmod` command associated with the `-exec` option, `{}` refers to the found file, and `\;` indicates the end of the command.

### read command

The `read` command assigns its argument with the line of input read and returns 0; if nothing is read in, it returns 1. Coupled with the `while` command, `read` can be used to process files or streams.

---

#### Example:

From a file containing a list of projects, select the lines in another data file with that line. Unlike `grep -f`, each project should be separated out.

---

```
while read project; do
    echo "$project:"
    grep "$project" usage.data
    echo
done < project.list
```

In the first example, `$project` successively has the value of lines in `project.list` (see file redirection after `done`); this value is written to the standard output stream. Using the `grep` command, the lines containing the value `$project` in the file `usage.data` are then written to the standard output stream. Last, an empty line is written to the standard output stream. As the `read` command returns 1 when there is no further input, it serves as the condition for the `while` construct.

Example:

Identifies directories with group- or other-write permissions

---

```
find / -type d \( -perm -g=w -o -perm -o=w \) | \
while read dir ; do
    echo "$dir is group- and/or world-writable"
done
```

Although this example could have been done with the `-exec` option of `find`, it illustrates a framework for doing more complex operations on the output of `find`. The `while` loop is similar to the last example except the input is from a pipe as opposed to file redirection. Thus, the utility to apply any piped input is demonstrated.

The option `-type` selects on the basis of file type, in this case, directory. Because the `-o`, or operator, has lower precedence than the implied `-a`, and operator, the `\(, \)` pair raises the precedence of the enclosed terms. The `-perm` terms select on write permission for the group and other. The resulting list consists of directories with group and/or other write permission bits set. Although the output only prints out an informative message, other combinations of tasks including mailing a system administrator or changing these permission bits are possible.

## In-line input redirection or here documents

Although the stdin for a command can be redirected from a file, it is useful in a script to have the data in the script as opposed to depending on a file. “Here” documents provide this functionality.

---

```
command <<flag  
line-1  
    :  
line-N  
flag
```

---

All the lines up to the line containing only *flag* are through stdin for *command*. Note: no spaces between “<<” and *flag*. All shell variables are expanded by default unless *flag* is preceded by a “\”.

Instead of reading from a file, this example uses a “here” document for input. An advantage to this is that it is not dependent on a data file in addition to the script. Any command that reads from the standard input stream can potentially have a “here” document as an alternative to an input file.

Example: selects total allocation and current usage for specific projects

```
cd /usr/accounting
while read account; do
    grep "$account:" allocations current
done <<EOF
PROJECT00000
PROJECT00000
PROJECT00000
PROJECT00000
PROJECT00000
EOF
```

The example selects lines from two files that contain the string **\$account**. Unlike the `grep` command also used, this snippet ensures that matched lines for the same string are kept together.

To shorten the file prefix as output by `grep`, the current working directory is changed. The effect is as if a file redirection had been used where the entries between “<<EOF” and “EOF” would have been the file contents. The `read` command successively assigns each entry to **account**.

## Functions

Functions are a way to group a series of commands to operate on different inputs. Functions can be likened to internal shell scripts as they can have command line arguments as well.

---

```
name () {  
    command-1  
    :  
    command-N  
}
```

---

Normally, the function returns to the caller when all commands have been executed. The return command can stop execution before the end of the function. The form “return *n*” gives an exit status of *n*; if *n* is omitted, the exit status is of the last command executed.

Functions must be defined before they are used in the script.

This example is spread across two pages. The first page contains a function that is used in the main script on the second page.

### Example:

The following example checks the ownership and permissions on files and directories. A function is defined to select the command that performs the “find” operation. Both commands take the same subset of command-line arguments.

```
generic_find () {
# Function to select and execute find or sfind as needed.
# Note: sfind only if there are no NFS mounts of the archive
SFIND=/bin/sfind
if test "$1" != "$ARCHIVE_HOME" ; then
    find "$@"
else
    if test "`rsh $ARCHIVE_HOST 'ls -d '$1`" = "$1" ; then
        rsh $ARCHIVE_HOST "$SFIND $@"
    fi
fi
}
```

The function first assigns to the variable **SFIND** the full path of the find command on the remote system. The if construct tests if the first argument is not equivalent to the value **\$ARCHIVE\_HOME**, the directory on the remote system. If true, the standard find command acts on all the command-line arguments. Otherwise, it checks that the first argument matches a directory listing on the remote system, value **\$ARCHIVE\_HOST**; if true, it executes the command **\$SFIND** on the system **\$ARCHIVE\_HOST**. By encapsulating the testing and execution within a function, maintenance is kept to a minimum.

```

for directory in $HOME $ARCHIVE_HOME; do
    echo "Checking for world accessible files/directories in: \
    $directory"

    generic_find $directory \( -type f -o -type d \) \
    \( -perm -o=r -o -perm -o=w -o -perm -o=x \) \
    -exec /bin/ls -ld {} \;

    echo
    echo "Checking for files/directories with invalid \
    owner/group in: $directory"

    generic_find $directory \( -nouser -o -nogroup \) \
    -exec /bin/ls -ld {} \;

    echo
    echo "Checking for files/directories not owned by $USER \
    in: $directory"

    generic_find $directory \
    \( ! -user $USER -o ! -group $USER \) \
    -exec /bin/ls -ld {} \;

    echo
done

```

The main script uses the `generic_find` function to check on properties of files and directories in the `$HOME` and `$ARCHIVE_HOME` paths. Use of the function hides the decision-making leading to a more uniform loop body.

In the first `generic_find` call, the file types are limited to ordinary files (`-type f`) and directories (`-type d`). Because the permission bits on links and other special file types may lead to false positives, this further culling was used. In addition, the check includes tests of any “other” permission bits being set. A long listing is generated for any matches for all the calls.

The second `generic_find` call uses the `-nouser` and `-nogroup` options to identify files without a defined user or group.

The third `generic_find` call checks for files that are not owned by user `$USER` or do not have the user’s group `$USER` as associated with the file. Here, the `!`, not operator, is used.

### . command

The `.` command executes its argument, a valid shell script, in the current shell. Thus, unlike executing the shell script by itself, it effects changes in the current shell as opposed to running in a child shell. The `.` command is useful for modifying the current process's environment or adding functions and aliases.

### exit command

The `exit` command terminates the current shell script at the point at which it is executed rather than allowing it to complete. If an optional argument  $n$  is included, it returns an exit status of  $n$ ; otherwise, it returns the exit status of the last command executed before `exit`. `exit` is useful for aborting the shell script if problems occur.

## Stream manipulation

The standard out and standard error streams (1 and 2) can be individually manipulated. This allows better control of redirection.

```
foo 2>&1 | more    # stdout and stderr piped into pager
# stderr only redirected to /dev/null (ignored)
grep $USER /.k5logins/*/k5login 2>/dev/null
bar 1>foo 2>snafu  # stdout and stderr to separate files
```

The first example shows how to tie standard out and error streams together. In this case, only standard out would be connected by the pipe to the less command. In a similar way, if standard out was redirected to a file, this construct would allow standard error to go to the same file.

The second example shows how to discard standard error (to the null device). The downside to this is that all errors, even relevant ones, are lost.

The last example redirects standard out (stream 1) to file **foo**, and standard error (stream 2) to file **snafu**.