

# Speaking UNIX: !\$#@\*%

## Learn even more command-line tricks and operators

Skill Level: Intermediate

[Adam T. Cormany \(acormany@yahoo.com\)](mailto:acormany@yahoo.com)

National Data Center Manager  
Scientific Games Corporation

30 Sep 2008

Get a better understanding of all those "strange" characters UNIX® users are typing. Learn how to use pipelines, redirections, operators, and more in UNIX.

So, you've worked on IBM® AIX® for a while now. You've learned a few of the basic commands to help you maneuver through a directory structure, create and modify files, see what processes are running, and maybe even administer users and the system. That's great, but you want to understand what the UNIX® administrators next to you are typing. It looks like a lot of commands interspersed with strange symbols. Learn what `|`, `>`, `>>`, `<`, `<<`, `[ [` and `]]`, and many more symbols mean in UNIX and Linux® as well as how to get the most out of operators such as `&&`, `||`, `<`, `<=`, and `!=`.

## Pipeline

If you're familiar with UNIX, the pipeline, or *pipe*, is an integral part of everyday processing. Originally developed by Malcolm McIlroy, the pipeline allows you to redirect the standard output (stdout) of one command to become the standard input (stdin) of the following command in a single chained execution. Using the pipeline isn't limited to one instance per execution. Quite often, the stdout of one command is used as stdin of the following command, and the subsequent stdout is redirected yet again as stdin to another command and so on.

For example, one of the first things most UNIX administrators do on their systems during troubleshooting or daily checks is look at processes running currently on the

system. [Listing 1](#) shows such a check.

### Listing 1. Example of a daily process check

```
# ps -ef

  UID      PID    PPID  C   STIME   TTY   TIME  CMD
  root         1        0   0   Jul 27   -    0:05
/etc/init
  root   53442   151674   0   Jul 27   -    0:00
/usr/sbin/syslogd
  root   57426        1   0   Jul 27   -    0:00
/usr/lib/errdemon
  root   61510        1   0   Jul 27   -   23:55
/usr/sbin/syncd 60
  root   65634        1   0   Jul 27   -    0:00
/usr/ccs/bin/shlap64
  root   82002   110652   0   Jul 27   -    0:24
/usr/lpp/X11/bin/X -x abx
-x dbe -x GLX -D /usr/lib/X11//rgb -T -force :0
-auth /var/dt/A:0-SfIdMa
  root   86102        1   0   Jul 27   -    0:00
/usr/lib/methods/ssa_daemon -l ssa0
  root  106538   151674   0   Jul 27   -    0:01
sendmail: accepting connections
  root  110652        1   0   Jul 27   -    0:00
/usr/dt/bin/dtlogin -daemon
  root  114754   118854   0   Jul 27   -   20:22 dtgreet
  root  118854   110652   0   Jul 27   -    0:00 dtlogin
<:0>
  root  131088        1   0   Jul 27   -    0:07
/usr/atria/etc/lockmgr
-a /var/adm/atria/almd -q 1024 -u 256 -f 256
  root  147584        1   0   Jul 27   -    0:01
/usr/sbin/cron
  root  155816   151674   0   Jul 27   -    0:04
/usr/sbin/portmap
  root  163968   151674   0   Jul 27   -    0:00
/usr/sbin/qdaemon
  root  168018   151674   0   Jul 27   -    0:00
/usr/sbin/inetd
  root  172116   151674   0   Jul 27   -    0:03
/usr/sbin/xntpd
  root  180314   151674   0   Jul 27   -    0:19
/usr/sbin/snmpmibd
  root  184414   151674   0   Jul 27   -    0:21
/usr/sbin/aixmibd
  root  188512   151674   0   Jul 27   -    0:20
/usr/sbin/hostmibd
  root  192608   151674   0   Jul 27   -    7:46
/usr/sbin/muxatmd
  root  196718   151674   0 11:00:27   -    0:00
/usr/sbin/rpc.mountd
  root  200818   151674   0   Jul 27   -    0:00
/usr/sbin/biod 6
  root  213108   151674   0   Jul 27   -    0:00
/usr/sbin/nfsd 3891
  root  221304   245894   0   Jul 27   -    0:05
/bin/nsrexecd
  daemon 225402   151674   0 11:00:27   -    0:00
/usr/sbin/rpc.statd
  root  229498   151674   0 11:00:27   -    0:00
/usr/sbin/rpc.lockd
  root  241794   151674   0   Jul 27   -    0:51
/usr/lib/netsvc/yp/ypbind
  root  245894        1   0   Jul 27   -    0:00
```

```

/bin/nsrexecd
  root 253960      1   0   Jul 27      -   0:00
./mflm_manager
  root 274568 151674   0   Jul 27      -   0:00
/usr/sbin/sshd -D
  root 282766      1   0   Jul 27    lft0  0:00
/usr/sbin/getty /dev/console
  root 290958      1   0   Jul 27      -   0:00
/usr/lpp/diagnostics/bin/diagd
  root 315646 151674   0   Jul 27      -   0:00
/usr/sbin/lpd
  root 319664      1   0   Jul 27      -   0:00
/usr/atria/etc/albd_server
  root 340144 168018   0 12:34:56      -   0:00
rpc.ttdbserver 100083 1
  root 376846 168018   0   Jul 30      -   0:00 rlogind
  cormany 409708 569522   0 19:29:27 pts/1  0:00 -ksh
  root 569522 168018   0 19:29:26      -   0:00 rlogind
  cormany 733188 409708   3 19:30:34 pts/1  0:00 ps -ef
  root 749668 168018   0   Jul 30      -   0:00 rlogind

```

The listing of the processes currently running on a system can be simple, as shown in [Listing 1](#); however, most production systems run several more processes that make the output of `ps` much longer. To shorten the list to what you're looking for, redirect the standard output of `ps -ef` using a pipeline to `grep` to search for exactly what you want to see. [Listing 2](#) shows the process list from [Listing 1](#) redirected to `grep` to search for the strings "rpc" and "ksh."

## Listing 2. Redirecting the process list to grep

```

# ps -ef | grep -E "rpc|ksh"

  root 196718 151674   0 11:00:27      -   0:00
/usr/sbin/rpc.mountd
  daemon 225402 151674   0 11:00:27      -   0:00
/usr/sbin/rpc.statd
  root 229498 151674   0 11:00:27      -   0:00
/usr/sbin/rpc.lockd
  root 340144 168018   0 12:34:56      -   0:00
rpc.ttdbserver 100083 1
  cormany 409708 569522   0 19:29:27 pts/1  0:00 -ksh
  cormany 733202 409708   0 19:52:20 pts/1  0:00 grep -E
rpc|ksh

```

Using the pipeline can be much more complicated when you redirect stdout to stdin several times. In the following example, the previous `ps` and `grep` example is expanded to pipeline the stdout to another `grep` to exclude any previous strings found that includes "grep" or "tttdbserver." When the final `grep` operation has finished, the stdout is redirected again using a pipeline to an `awk` statement to print any of the processes found with a process identifier (PID) larger than 200,000:

```

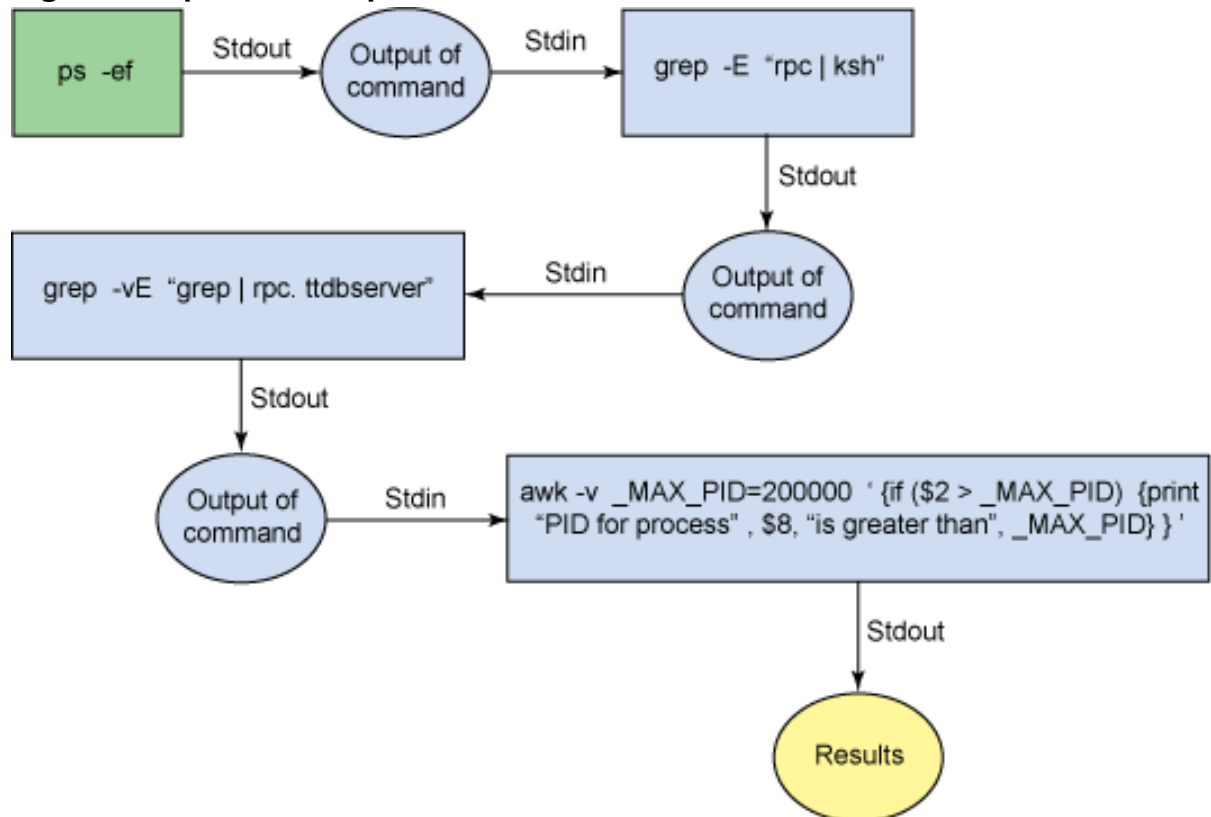
# ps -ef | grep -E "rpc|ksh" | grep -vE
"grep|rpc.ttdbserver" |
  awk -v _MAX_PID=200000 '{if ($2 > _MAX_PID) {print "PID
for
  process", $8, "is greater than", _MAX_PID}}'

```

```
PID for process /usr/sbin/rpc.statd is greater than 200000
PID for process /usr/sbin/rpc.lockd is greater than 200000
PID for process -ksh is greater than 200000
```

Figure 1 provides a graphical representation of the command's stdout redirecting to stdin for the subsequent command.

**Figure 1. Pipeline example**



## Data redirection with >, >>, <, and <<

Another important aspect of executing commands from the command-line interface (CLI) is the ability to write various outputs to a device or to read input into a command from another device. To write the output of a command, append the greater-than symbol (> or >>) and the target file name or device desired after the command to be executed. If the target file doesn't exist and you have Write permissions to the target directory, > and >> create the file with permissions of your umask and write the command's output to the newly created file. If, however, the file does exist, > attempts to open the file and overwrite the entire contents. If you would rather append to the file, simply use >>. Think of it as the flow of output data moving from the command on the left moving to the destination file on the right (that is, <cmd> -> <output> -> <file>).

The following example executes the `ps -ef` sample shown in the section, "[Pipeline](#)," and redirects the output to a file named `ps_out`:

```
# ps -ef | grep -E "rpc|ksh" > ps_out
```

The following code executes the earlier extended pipeline example and redirects the output to the same file—`ps_out`—but appends to the current data:

```
# ps -ef | grep -E "rpc|ksh" | grep -vE
"grep|rpc.ttdbserver" |
awk -v _MAX_PID=200000 '{if ($2 > _MAX_PID) {print "PID
for
process", $8, "is greater than", _MAX_PID}}' >> ps_out
```

[Listing 3](#) shows the output from the last two redirections.

### Listing 3. Output from subsequent redirections

```
# cat ps_out

    root  196718  151674    0 11:00:27      -   0:00
/usr/sbin/rpc.mountd
    daemon 225402  151674    0 11:00:27      -   0:00
/usr/sbin/rpc.statd
    root  229498  151674    0 11:00:27      -   0:00
/usr/sbin/rpc.lockd
    root  340144  168018    0 12:34:56      -   0:00
rpc.ttdbserver 100083 1
    cormany 409708  569522    0 19:29:27 pts/1 0:00 -ksh
    cormany 733202  409708    0 19:52:20 pts/1 0:00 grep -E
rpc|ksh
PID for process /usr/sbin/rpc.statd is greater than 200000
PID for process /usr/sbin/rpc.lockd is greater than 200000
PID for process -ksh is greater than 200000
```

When redirecting output with `>` alone, only the stdout of the command is redirected. Keep in mind that with computing, there is stdout as well as stderr: The former is represented as 1, while stderr is 2. Redirecting output in UNIX is no different. Simply place the desired output type before the `>` (for example, `1>`, `2>`) to tell the shell where to route the output.

[Listing 4](#) attempts to list files `fileA.tar.bz2` and `fileC.tar.bz2`. Unfortunately, as shown in the first command (`ls`), `fileC.tar.bz2` doesn't exist. Thankfully, we remembered to separate stdout into `ls.out` and stderr into `ls.err`.

### Listing 4. Listing the files `fileA.tar.bz2` and `fileC.tar.bz2`

```
# ls
fileA.tar.bz2  fileAA.tar.bz2  fileB.tar.bz2
fileBB.tar.bz2
```

```
# ls fileA.tar.bz2 fileC.tar.bz2 1> ls.out 2> ls.err

# cat ls.out
fileA.tar.bz2

# cat ls.err
ls: 0653-341 The file fileC.tar.bz2 does not exist.
```

The same rules apply in AIX with `>` and `>>` on stdout and stderr. For example, the same output files can be used for future tests, as [Listing 5](#) shows.

### Listing 5. Using output files for future tests

```
# ls fileB.tar.bz2 fileD.tar.bz2 1>> ls.out 2>> ls.err

# cat ls.out
fileA.tar.bz2
fileB.tar.bz2

# cat ls.err
ls: 0653-341 The file fileC.tar.bz2 does not exist.
ls: 0653-341 The file fileD.tar.bz2 does not exist.
```

There are times when you may need to have both stdout and stderr written to the same file or device. You can do this in either of two ways. The first method is to direct `1>` and `2>` to the same file:

```
# ls fileA.tar.bz2 fileC.tar.bz2 1> ls.out 2> ls.out

# cat ls.out
fileA.tar.bz2
ls: 0653-341 The file fileC.tar.bz2 does not exist.
```

The second method is a simpler and quicker way to accomplish the same thing and is used more frequently by experienced UNIX users:

```
# ls fileA.tar.bz2 fileC.tar.bz2 > ls.out 2>&1

# cat ls.out
fileA.tar.bz2
ls: 0653-341 The file fileC.tar.bz2 does not exist.
```

Let's break the statement down. First, `ls fileA.tar.bz2 fileC.tar.bz2` is executed. The stdout is redirected to `ls.out` with `> ls.out`, and stderr is redirected to the same file to which stdout is redirected (`ls.out`) with `2>&1`.

Remember that you can redirect output to files as well as other devices. You can redirect data to printers, floppy disks, Terminal Types (TTYs), and various other devices. For example, if you wanted to send a message to a single user on all sessions (or TTYs), you could just loop through `who` and redirect a message to the TTYs if you have adequate permissions, as shown in [Listing 6](#).

## Listing 6. Redirecting a message to a TTY

```
# for _TTY in `who | grep "cormany" | awk '{print $2}'`
> do
>   _TTY="/dev/${_TTY}"
>   echo "Sending message to cormany on ${_TTY}"
>   echo "Test Message to cormany@${_TTY}" > ${_TTY}
> done

Sending message to cormany on /dev/pts/13
Test Message to cormany@/dev/pts/13
Sending message to cormany on /dev/pts/14
```

## Stdin, not stdout

Although using `>` and `>>` seems a relatively easy concept for most to pick up, it's common for others to have difficulties using the less-than symbols (`<` and `<<`). When thinking of `>` and `>>`, it's easiest to visualize them as the flow of output data moving from the command on the left to the destination file on the right. The same applies to `<` and `<<`. Using `<`, you essentially execute a command with stdin already supplied. Think of it as the data already provided supplied to the command on the left of the data as stdin (that is, `<cmd> <- <data>`).

For example, say you want to send an e-mail of an ASCII text file to another user. You could use a pipeline to redirect the stdout of `cat` to stdin of `mail` (that is, `cat mail_file.out | mail -s "Here's your E-mail!" acormany@yahoo.com`), or you could redirect the contents of the file to become stdin for the `mail` command:

```
# mail -s "Here's your E-mail!" acormany@yahoo.com <
mail_file.out
```

Using `<<`, also known as a *here-document*, can save some formatting time and is easier on the processing time of the command execution. By using `<<`, the string of text is directed to the command to execute as stdin, but you can continue to enter information until the termination identifier has been reached. Simply type the command following `<<` and the termination identifier, type anything you want, and end it with the termination identifier on a new line. Using the here-document allows you to preserve whitespace, new lines, and so on.

For example, rather than typing five `echo` statements that UNIX would have to process individually:

```
# echo "Line 1"
Line 1

# echo "Line 2"
```

```
Line 2

# echo "Line 3"
Line 3

# echo "Line 4"
Line 4

# echo "Line 5"
Line 5
```

you could use the following code to replace the multi-`echo` statement, and UNIX would only need to process a single execution:

```
# cat << EOF
> Line 1
> Line 2
> Line 3
> Line 4
> Line 5
> EOF

Line 1
Line 2
Line 3
Line 4
Line 5
```

To allow tabs to make everything look a bit neater in the shell script, simply place a hyphen (-) between the `<<` and the termination identifier:

```
# cat <<- ATC
> Line 1
> Line 2
> Line 3
> Line 4
> Line 5
> ATC

Line 1
Line 2
Line 3
Line 4
Line 5
```

[Listing 7](#) provides an example of how to combine a few items discussed in this article so far.

### Listing 7. Combining CLI

```
# cat redirect_example

#!/usr/bin/ksh

cat <<- ATC | sed "s/^/Redirect Example => /g" >> atc.out
This is an example of how to redirect
```



```
stdout to a file as well as pipe stdout into stdin
of another command (i.e. sed), all done inside
a here-document.
```

```
Cool eh?
ATC
```

Now let's see what the script looks like with the redirection and pipeline.

```
# ./redirect_example

# cat atc.out
Redirect Example => This is an example of how to redirect
Redirect Example => stdout to a file as well as pipe
stdout into stdin
Redirect Example => of another command (i.e. sed), all
done inside
Redirect Example => a here-document.
Redirect Example =>
Redirect Example => Cool eh?
```

## Subshells

Sometimes, you need to execute several commands together. For example, if you want to perform a specific action in a different directory, you could use the code in [Listing 8](#).

### Listing 8. Execute several commands at the same time

```
# pwd
/home/cormany

# cd testdir

# tar -cf ls_output.tar ls.out?

# pwd
/home/cormany/testdir
```

This works, but note that after the execution of these steps, you're no longer in your original directory. By placing the commands into their own subshell, they execute as a single instance of the subshell. [Listing 9](#) shows the same idea executed using a subshell.

### Listing 9. Execute several commands at the same time using a subshell

```
# pwd
/home/cormany

# (cd testdir ; tar -cf ls_output.tar ls.out?)

# pwd
```

```
/home/cormany
```

## The test command, [ ], and [[ ]]

When writing a shell script or programming in any modern language, the ability to evaluate expressions or values is essential to competent programs. UNIX has it covered as always with the `test` command. As the `test` man page states, the `test` command evaluates expression parameters and, if the expression value is True, returns a zero (True) exit value. For more information on the definition of `test` and all the available conditions, see the `test` man page.

To use the `test` command, simply provide the command with the appropriate flag and file name. When `test` has evaluated the expression, you're returned to a command prompt, where you can verify the return code, as shown in [Listing 10](#).

### Listing 10. Verify return code

```
# ls -l
-rwxr-xr-x    1 cormany  atc              786 Feb 22 16:11
check_file
-rw-r--r--    1 cormany  atc              0 Aug 04 20:57
emptyfile

# test -f emptyfile
# echo $?
0

# test -f badfilename
# echo $?
1
```

As stated in the definition, `test` returns a zero exit value if the expression value was True or a non-zero exit value (that is, 1). In [Listing 10](#), the file *emptyfile* was found, so `test` returned 0; the file *badfilename* was not found, so 1 was returned.

Another way to use the `test` command is to place the expression to evaluate within single brackets (`[ ]`). Using the `test` command or replacing it with `[ ]` returns the same value, as they are identical executions:

```
# [ -f emptyfile ]
# echo $?
0

# [ -f badfilename ]
# echo $?
1
```

Using single brackets (`[ ]`) versus double brackets (`[[ ]]`) is a personal preference and really depends on how you've been taught commands and shell

scripting. But keep in mind that there are some differences between the two evaluations. Although `[ ]` and `[[ ]]` use the same test operators during evaluation, they use different logical operators.

## Operators

In ksh, the default shell used in AIX, as well as other shells used in UNIX and Linux, it's important to know how to use test, logical, and substitution operators.

### Test operators

When writing shell scripts, test operators are crucial to error checking and for checking the status of files. The following test operators are just a few that you can use in ksh as well as other standard UNIX shells:

- **-d <file>**: <file> is a directory
- **-e <file>**: <file> exists
- **-f <file>**: <file> is a regular file
- **-n <string>**: <string> is not NULL
- **-r <file>**: The user has Read permissions to <file>
- **-s <file>**: <file> size is greater than 0
- **-w <file>**: The user has Write permissions to <file>
- **-x <file>**: The user has Execute permissions to <file>
- **-z <string>**: <string> is null
- **-L <file>**: <file> is a symbolic link

Remember, in UNIX directories, devices, symbolic links, and other objects are all files, so the test operators shown above will work with every type of file.

Everyone has an individual style of shell scripting. Whether they use `[[ ]]` or `[ ]` in test statements, the above test operators will function the same. This article uses `[ ]`. [Listing 11](#) shows how you can use a few of the test operators listed above.

### Listing 11. Using test operators

```
#!/usr/bin/ksh

while true
do
    echo "\nEnter file to check:  \c"
    read _FNAME
```

```

if [[ ! -e "${_FNAME}" ]]
then
    echo "Unable to find file '${_FNAME}'"
    continue
fi

if [[ -f "${_FNAME}" ]]
then
    echo "${_FNAME} is a file."
elif [[ -d "${_FNAME}" ]]
then
    echo "${_FNAME} is a directory."
elif [[ -L "${_FNAME}" ]]
then
    echo "${_FNAME} is a symbolic link."
else
    echo "Unable to determine file type for '${_FNAME}'"
fi

[[ -r "${_FNAME}" ]] && echo "User ${USER} can read
'${_FNAME}'"
[[ -w "${_FNAME}" ]] && echo "User ${USER} can write to
'${_FNAME}'"
[[ -x "${_FNAME}" ]] && echo "User ${USER} can execute
'${_FNAME}'"

if [[ -s "${_FNAME}" ]]
then
    echo "${_FNAME} is NOT empty."
else
    echo "${_FNAME} is empty."
fi
done

```

Executing the code in Listing 11 and checking a few file names produces the output shown in [Listing 12](#).

### Listing 12. Output from executing the test operators

```

# ls -l
-rwxr-xr-x    1 cormany  atc                786 Feb 22 16:11
check_file
-rw-r--r--    1 cormany  atc                0 Aug 04 20:57
emptyfile

# ./check_file

Enter file to check: badfilename
Unable to find file 'badfilename'

Enter file to check: check_file
check_file is a file.
User cormany can read 'check_file'
User cormany can write to 'check_file'
User cormany can execute 'check_file'
check_file is NOT empty.

Enter file to check: emptyfile
emptyfile is a file.
User cormany can read 'emptyfile'
User cormany can write to 'emptyfile'
emptyfile is empty.

```

To learn more about test operators and to see a complete listing of test operators, execute `man test`.

## Logical operators

Another important set of operators in UNIX is the logical operators. Like in most modern programming languages, the `AND` and `OR` statements are necessary for definitive conditional evaluations of expressions or their values.

If you've read any of my previous articles (see [Resources](#)), you'll notice that I favor logical operators over writing several lines of code. This keeps the scripts clean and easy to manage. One of the first things I do when writing a script is to write the `exit_msg()` function:

```
exit_msg() {
    [[ $# -gt 1 ]] && echo "${0##*/} (${1}) - ${2}"
    exit ${1:-0}
}
```

rather than having ugly and bloated code like that shown in [Listing 13](#).

### Listing 13. The alternative to using the `exit_msg()` function and clean logical operators

```
#!/usr/bin/ksh

if [[ -n ${_NUM1} ]]
then
    unset _NUM1
fi

if [[ -n ${_NUM2} ]]
then
    unset _NUM2
fi

while [[ -z ${_NUM1} ]] || [[ -z ${_NUM2} ]]
do
    echo "Enter 2 sets of numbers: \c"
    read _NUM1 _NUM2
done

echo "Enter file to log results to: \c"
read _FNAME

if [[ ! -e "${_FNAME}" ]]
then
    echo "File '${_FNAME}' doesn't exist. A new log will be
    created."
fi

touch "${_FNAME}"

if [[ ! -w "${_FNAME}" ]]
then
    echo "Unable to write to file '${_FNAME}'"
    exit 1
```

```

fi

expr ${_NUM1} \/ 1 > /dev/null 2>&1
if [[ $? -ne 0 ]]
then
    echo "Number '${_NUM1}' is not numeric."
    exit 2
fi

expr ${_NUM2} \/ 1 > /dev/null 2>&1
if [[ $? -ne 0 ]]
then
    echo "Number '${_NUM2}' is not numeric."
    exit 2
fi

echo "${_NUM1},${_NUM2}" >> "${_FNAME}"

```

By using a simple function like `exit_msg()` and a few logical operators, the script could be condensed into the better-looking and easier-to-understand program shown in [Listing 14](#).

#### Listing 14. Cleaner version of a script using functions and logical operators

```

#!/usr/bin/ksh

exit_msg() {
    [[ $# -gt 1 ]] && echo "${0##*/} (${1}) - ${2}"
    exit ${1:-0}
}

[[ -n ${_NUM1} ]] && unset _NUM1
[[ -n ${_NUM2} ]] && unset _NUM2

while [[ -z ${_NUM1} ]] || [[ -z ${_NUM2} ]]
do
    echo "Enter 2 sets of numbers: \c"
    read _NUM1 _NUM2
done

echo "Enter file to log results to: \c"
read _FNAME

[[ ! -e "${_FNAME}" ]] && echo
    "File '${_FNAME}' doesn't exist. A new log will be
    created."

touch "${_FNAME}"

[[ ! -w "${_FNAME}" ]] && exit_msg 1
    "Unable to write to file '${_FNAME}'"

expr ${_NUM1} \/ 1 > /dev/null 2>&1
[[ $? -ne 0 ]] && exit_msg 2 "Number '${_NUM1}' is not
numeric."

expr ${_NUM2} \/ 1 > /dev/null 2>&1
[[ $? -ne 0 ]] && exit_msg 2 "Number '${_NUM2}' is not
numeric."

echo "${_NUM1},${_NUM2}" >> "${_FNAME}"

```

The previous examples focused more on the AND (&&) and OR (||) logical operators. In addition to these, you can use the AND (-a) and OR (-o) operators as discussed in the section describing [ ] versus [[ ]]. If using the `test` command or single brackets ([ ]), use -a and -o to evaluate the expression. If, however, you use double brackets ([[ ]]), use && and ||:

```
# [[ "Paul" != "Xander" && 2 -gt 0 ]]
# echo $?
0

# [ "Paul" != "Xander" -a 2 -gt 0 ]
# echo $?
0
```

## Comparison test operators

Another set of test operators is called *comparison test operators*. Like the previous set of test operators, comparison test operators are a handy way to perform error checking or to test values against another value. The previous test operators were used mostly on files or to see if a variable was defined, but the comparison test operators are used more on strings and numeric values. This can be useful when checking dates, file sizes, if one string is the same as another string, and so on.

The comparison test operators are:

- **<fileA> -nt <fileB>**: fileA is newer than fileB
- **<fileA> -ot <fileB>**: fileA is older than fileB
- **<fileA> -ef <fileB>**: fileA and fileB point to the same file
- **<string> = <pattern>**: string matches pattern
- **<string> != <pattern>**: string does not match pattern
- **<stringA> < <stringB>**: stringA comes before stringB in dictionary order
- **<stringA> > <stringB>**: stringA comes after stringB in dictionary order
- **<exprA> -eq <exprB>**: expressionA is equal to expressionB
- **<exprA> -ne <exprB>**: expressionA is not equal to expressionB
- **<exprA> -lt <exprB>**: expressionA is less than expressionB
- **<exprA> -gt <exprB>**: expressionA is greater than expressionB
- **<exprA> -le <exprB>**: expressionA is less than or equal to expressionB

- **<exprA> -ge <exprB>:** expressionA is greater than or equal to expressionB

You use the same format on comparison test operators as other operators. You can use either `test`, `[ ]`, or `[[ ]]`. [Listing 15](#), [Listing 16](#), and [Listing 17](#) display how you can use numeric, string, and file comparisons, respectively.

### Listing 15. Numeric comparisons

```
# ls -l *.file
-rw-r--r-- 1 cormany atc 21 Feb 22 2006
Pauls.file
-rw-r--r-- 1 cormany atc 22 Aug 04 20:57
Xanders.file

# [[ "Pauls.file" -ot "Xanders.file" ]]
# echo $?
0
```

### Listing 16. String comparison

```
# _PSIZE=`ls -l Pauls.file | awk '{print $5}'`
# _XSIZE=`ls -l Xanders.file | awk '{print $5}'`
# [[ ${_PSIZE} -lt ${_XSIZE} ]]
# echo $?
0
```

### Listing 17. File comparison

```
# [[ "cat" = "dog" ]]
# echo $?
1
```

## Substitution operators

It's easy to forget to define a variable or assign a value to it when a script grows or you haven't touched the script for years and need to add to it. Other times, it would be handy to tell users that a value is set or set up some defaults for your users. Substitution operators are a great address to these problems:

- **\${var-value}:** If `<var>` exists, return `<var>`'s value. If `<var>` doesn't exist, return `<value>`.
- **\${var=value}:** If `<var>` exists, return `<var>`'s value. If `<var>` doesn't exist, set `<var>` to `<value>` and return `<value>`.



- **`${var+value}`**: If `<var>` exists, return `<value>`. If `<var>` doesn't exist, return NULL.
- **`${var?value}`**: If `<var>` exists, return `<var>`'s value. If `<var>` doesn't exist, exit the command or script and display the error message set with `<value>`. If `<value>` isn't set, a default error message of "Parameter null or not set" is displayed.
- **`${var:-value}`**: If `<var>` exists and isn't NULL, return `<var>`'s value. If `<var>` doesn't exist or is NULL, return `<value>`.
- **`${var:=value}`**: If `<var>` exists and isn't NULL, return `<var>`'s value. If `<var>` doesn't exist or is NULL, set `<var>` to `<value>` and return `<value>`.
- **`${var:+value}`**: If `<var>` exists and isn't NULL, return `<value>`. If `<var>` doesn't exist or is NULL, return NULL.
- **`${var:?value}`**: If `<var>` exists and isn't NULL, return `<var>`'s value. If `<var>` doesn't exist or is NULL, exit the command or script and display the error message set with `<value>`. If `<value>` isn't set, a default error message of "Parameter null or not set" is displayed.

Note the subtle difference between the first group of four definitions and the second set of four. The last set includes a colon (:) between the variable name and the substitution operator, which adds the check to see if the variable is NULL, as well. Another important note to think about when trying to assign values to variables with substitution operators is that assigning a value to a variable has the same rules as defining a variable normally from the command line or a script. Protected reserved variables cannot be overwritten with a new value (for example, `$1`, `$2`, `$3`).

[Listing 18](#) provides an example of how the variables work. Note that you can combine several substitution operators, as shown in the last line of the script.

### Listing 18. Using substitution operators

```
# cat subops_examples

#!/usr/bin/ksh

_ARG1="${1}"
echo "Test 1A: The 1st argument is ${_ARG1-'ATC'}"
echo "Test 1B: The 1st argument is ${_ARG1:-'ATC'}"

_ARG2="${2}"
echo "Test 2A: The 2nd argument is ${_ARG2-'AMDC'}"
echo "Test 2B: The 2nd argument is ${_ARG2:-'AMDC'}"

_ARG3="${3}"
echo "Test 3A: The 3rd argument is ${_ARG3-'PAC'}"
echo "Test 3B: The 3rd argument is ${_ARG3:= 'PAC'}"

_ARG4="${4}"
```

```

echo "Test 4A: ${4:+'The 4th argument was supplied'}"

echo "Test 5: If the 4th argument was provided, the value
would be
    ${4:?'The 4th argument was not supplied.'}. Otherwise,
we will not
    see this message and get an error instead."

_ARG8="${8}"
echo "${_ARG8:=${7:-${6:-${5:-No Arguments were supplied
after the 4th}}}}}"

```

[Listing 19](#) shows how to execute the script with no argument supplied.

### Listing 19. Execute the script without arguments

```

# ./subops_examples
Test 1A: The 1st argument is
Test 1B: The 1st argument is ATC
Test 2A: The 2nd argument is
Test 2B: The 2nd argument is AMDC
Test 3A: The 3rd argument is
Test 3B: The 3rd argument is PAC
Test 4A:
./subops_examples[18]: 4: The 4th argument was not
supplied.

```

[Listing 20](#) shows what happens when executing the script with only three arguments.

### Listing 20. Execute the script with three arguments

```

# ./subops_examples arg1 arg2 arg3
Test 1A: The 1st argument is arg1
Test 1B: The 1st argument is arg1
Test 2A: The 2nd argument is arg2
Test 2B: The 2nd argument is arg2
Test 3A: The 3rd argument is arg3
Test 3B: The 3rd argument is arg3
Test 4A:
./subops_examples[18]: 4: The 4th argument was not
supplied.

```

[Listing 21](#) shows what happens when you supply only four arguments.

### Listing 21. Execute the script with four arguments

```

# ./subops_examples arg1 arg2 arg3 arg4
Test 1A: The 1st argument is arg1
Test 1B: The 1st argument is arg1
Test 2A: The 2nd argument is arg2
Test 2B: The 2nd argument is arg2
Test 3A: The 3rd argument is arg3
Test 3B: The 3rd argument is arg3
Test 4A: The 4th argument was supplied
Test 5: If the 4th argument was provided, the value would
be
    arg4. Otherwise, we will not see this message and get

```

```
an
    error instead.
No Arguments were supplied after the 4th
```

[Listing 22](#) shows all five arguments supplied.

### Listing 22. Execute the script with all five arguments

```
# ./subops_examples arg1 arg2 arg3 arg4 arg5
Test 1A: The 1st argument is arg1
Test 1B: The 1st argument is arg1
Test 2A: The 2nd argument is arg2
Test 2B: The 2nd argument is arg2
Test 3A: The 3rd argument is arg3
Test 3B: The 3rd argument is arg3
Test 4A: The 4th argument was supplied
Test 5: If the 4th argument was provided, the value would
be
    arg4. Otherwise, we will not see this message and get
an
    error instead.
arg5
```

[Listing 23](#) shows seven arguments supplied. Note how arguments 5 and 6 were ignored, because seven arguments were provided.

### Listing 23. Execute the script with seven arguments

```
# ./subops_examples arg1 arg2 arg3 arg4 arg5 arg6 arg7
Test 1A: The 1st argument is arg1
Test 1B: The 1st argument is arg1
Test 2A: The 2nd argument is arg2
Test 2B: The 2nd argument is arg2
Test 3A: The 3rd argument is arg3
Test 3B: The 3rd argument is arg3
Test 4A: The 4th argument was supplied
Test 5: If the 4th argument was provided, the value would
be
    arg4. Otherwise, we will not see this message and get
an
    error instead.
arg7
```

## Conclusion

After reading this article, you should have a better understanding of all those "strange" characters UNIX users are typing. Knowing how to redirect data as stdin or stdout, how to use the pipe, and how to use operators in UNIX helps you write more powerful scripts with better error trapping and cleaner logic. Good luck!

# Resources

## Learn

- [Speaking UNIX](#): Check out other parts in this series.
- [test command](#): See IBM's commands reference for the UNIX `test` command.
- [Redirecting output to here-documents](#): See IBM's infocenter information on this type of redirection.
- [Input and output redirection](#): See IBM's infocenter entry on input and output redirection.
- [Wikipedia's pipeline entry](#): Read Wikipedia's excellent entry on pipelines in the UNIX environment.
- [Wikipedia's definition of the UNIX test command](#): Read Wikipedia's entry on the UNIX `test` command.
- [The AIX and UNIX developerWorks zone](#) provides a wealth of information relating to all aspects of AIX systems administration and expanding your UNIX skills.
- [New to AIX and UNIX?](#) Visit the New to AIX and UNIX page to learn more.
- [developerWorks technical events and webcasts](#): Stay current with developerWorks technical events and webcasts.
- [AIX Wiki](#): Visit this collaborative environment for technical information related to AIX.
- [Podcasts](#): Tune in and catch up with IBM technical experts.

## Get products and technologies

- [IBM trial software](#): Build your next development project with software for download directly from developerWorks.

## Discuss

- Participate in the AIX and UNIX forums:
  - [AIX Forum](#)
  - [AIX Forum for developers](#)
  - [Cluster Systems Management](#)
  - [IBM Support Assistant Forum](#)
  - [Performance Tools Forum](#)
  - [Virtualization Forum](#)

- [More AIX and UNIX forums](#)

## About the author

Adam T. Cormany

Adam Cormany is currently the manager of the National Data Center, but he has also been a UNIX systems engineer, a UNIX administrator, and operations manager for Scientific Games Corporation. Adam has worked extensively with AIX as well as in Solaris and Red Hat Linux administration for more than 10 years. He is an IBM eServer®-Certified Specialist in pSeries® AIX System Administration. In addition to administration, Adam has extensive knowledge of shell scripting in Bash, CSH, and KSH as well as programming in C, PHP, and Perl. You can reach Adam at [acormany@yahoo.com](mailto:acormany@yahoo.com).

## Trademarks

IBM, AIX, eServer, and pSeries are registered trademarks of International Business Machines in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.