**Speaking UNIX: Hello, shell**

Source: <http://www.ibm.com/developerworks/views/aix/libraryview.jsp?search_by=Speaking+UNIX+Part>

Resource: http://www.ibm.com/developerworks/aix/library/au-unix-find.html

One of the most novel and differentiating features of a UNIX® system is its *command line*. With just a few keystrokes, including a bit of "glue", you can use the command line to combine the finite set of UNIX utilities into innumerable, impromptu data transforms.

For example, to find the list of unique filenames in the folder hierarchy rooted at the current working directory, you can type the following at your shell prompt:

|  |
| --- |
| **find . -type f -print | sort | uniq** |

This command line combines three separate utilities:

* find plumbs the depths of the named directory -- in this case, the file system starting at . or *dot* (shorthand for the current working directory) -- and emits the names of all entries that match the given criteria. Here, -type f directs find to discover only plain files.
* sort, as its name implies, processes a list and emits a new list that's sorted alphabetically.
* uniq (pronounced "unique") scans a list, comparing adjacent elements in the list and removing any duplicates. For instance, suppose you have this list:

**Listing 1. Example list**

|  |
| --- |
| Groucho  Groucho  Chico  Chico  Groucho  Harpo  Zeppo  Zeppo |

uniq reduces the list to the following:

**Listing 2. uniq command**

|  |
| --- |
| Groucho  Chico  Groucho  Harpo  Zeppo |

However, if the original list of Marx Brothers is sorted first (reordering all occurrences of a name into a continuous run), running uniq yields this result:

**Listing 3. Running uniq**

|  |
| --- |
| Chico  Groucho  Harpo  Zeppo |

To learn more about the extensive features of find, sort, and uniq, refer to each utility's man page on your UNIX system.

Data in, data out, data all about

Used independently, find always takes the contents of the file system as its input data. However, both sort and uniq require data entry or input from *the standard input device* (stdin). Most often, you provide stdin using the keyboard: You type the data you want sorted on a series of lines, for example.

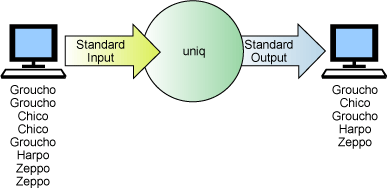
By default, find prints results on the *standard output device* (stdout), which is usually your terminal window. Both sort and uniq print outcomes to stdout.

To demonstrate stdin and stdout, type the following text in your terminal window (assume that the leading percent sign (%) is your shell prompt):

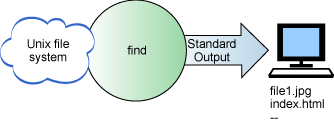
**Listing 4. stdin and stdout**

|  |
| --- |
| % **sort**  **mustache**  **horn**  **hat**  **Control-D** |

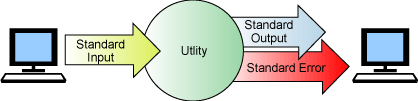
sort reads the three lines you typed from stdin, sorts them, and writes the result to stdout. [Figure 1](http://www.ibm.com/developerworks/aix/library/au-unix-commandline/index.html#fig1) presents a conceptual picture of running sort, and most UNIX command-line utilities, from the command line.

**Figure 1. A typical UNIX command-line utility reads from stdin and writes to stdout**  


Some utilities, such as find, don't read from stdin. Instead, they read the data they should process from system resources, such as the file system or the system kernel, and write results to stdout. To visualize how find works, look at [Figure 2](http://www.ibm.com/developerworks/aix/library/au-unix-commandline/index.html#fig2) below.

**Figure 2. Some utilities read data from system resources and write results to stdout**  


In addition to using stdin and stdout, UNIX commands can emit error messages to a special outlet that's set aside (by convention, not mandate) for diagnostics. The outlet is called the *standard error device* (usually referred to as stderr). [Figure 3](http://www.ibm.com/developerworks/aix/library/au-unix-commandline/index.html#fig3) illustrates a simple command line running a utility.

**Figure 3. UNIX commands emit errors to a special channel, standard error**  


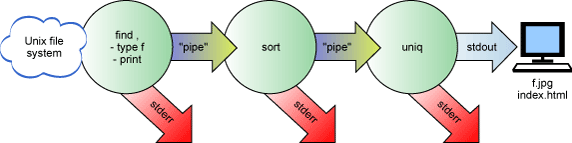
As shown in [Figure 3](http://www.ibm.com/developerworks/aix/library/au-unix-commandline/index.html#fig3), most UNIX commands read input from the terminal, send results to the terminal, and print errors to the terminal. By default, and unless you specify otherwise, your terminal is the source of data for stdin and the destination for both stdout and stderr.

Routing traffic to and fro

However, you can change the source of stdin and the eventual destinations of both stdout and stderr. You can force stdin to read from a text file, a device (say, a probe connected to the computer), or a network connection. Comparably, you can send output to a file, a device, or aconnection. In UNIX, where everything is a file, one source or destination is just as easy to consume or produce as another.

Changing the source and destination of a process's data is referred to as *redirection*. You redirect stdin to read data from a file or other source; and you can redirect stdout and stderr (separately) to write data somewhere other than the terminal window. In many cases, as in the original find command shown earlier, you can also redirect utilities to consume and produce data from and for other utilities. That is the purpose of the *pipe* (|). In a command, you can daisy-chain processes together using pipes, sending the data of one command to the next command, just like segments of copper pipe route water from your water heater to your sink.

[Figure 4](http://www.ibm.com/developerworks/aix/library/au-unix-commandline/index.html#fig4) shows a conceptualization of the find . -type f -print | sort | uniq command.

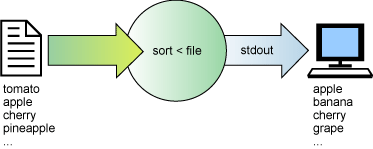
**Figure 4. A conceptual model of three utilities linked by pipes**  


The stdout of find becomes the stdin of uniq; in turn, the stdout of uniq becomes the stdin for sort. Finally, sort prints the results to its standard output device, which remains connected to the terminal. The stderr of the commands wasn't redirected, so all three utilities print error messages to the terminal. (Error messages from the three utilities are intermixed, but the order of the messages will be correct.)

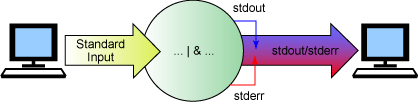
If necessary, you can extend the pipeline shown above further and redirect the output of uniq to yet another utility. Just tack on another pipe to extend the transform further. For instance, you can append | less to paginate the output using less, or you can add | wc -l to find the number of unique filenames. (wc is an acronym for *word count*; wc can count characters, words, and lines.)

Alternatively, you can use > to save the output of the entire sequence to a file (destroying the existing contents of the file, if any). You can also use >> to *append* the results to an existing file (creating the file if it doesn't exist).

Another helpful redirection is <. [Figure 5](http://www.ibm.com/developerworks/aix/library/au-unix-commandline/index.html#fig5) shows how stdin can be redirected to read from a file. The command sort reads a list of words from the named file and alphabetizes them.

**Figure 5. Redirecting standard input to read from a file**  


Often, you'll want to capture stdout and stderr. For example, if you're running a large data-mining task, you might want to review the interim output and any errors that occurred during execution. You can use variants of the redirection syntax to do just that: |&, >&, >>&, pipe, create, and append stdout and stderr simultaneously, respectively. [Figure 6](http://www.ibm.com/developerworks/aix/library/au-unix-commandline/index.html#fig6) illustrates how stdout and stderr are combined into one output stream.

**Figure 6. Combining the standard output and standard error devices**  


Introducing the Z shell

Most modern UNIX shells -- including the Bourne shell (bash) and the Korn shell (ksh) -- support the redirections mentioned here, although the specific syntax required by both of those shells might differ slightly. (Check your shell's documentation for specifics.).

Most of the redirection operators have been consistent features of all UNIX shells for at least 25 years. However, most of those shells have failed to break new ground and explore new ways to apply redirection. For instance, most shells can only redirect input from a single file, and you must use a utility like tee to output to more than one destination. (Like the tee junction used by plumbers, tee has one input and two outputs.) Here's an example using bash as the shell (the command-line interpreter):

**Listing 5. bash example**

|  |
| --- |
| bash$ ls  tellme  bash$ **cat tellme**  echo Your current login, working directory, and system are...  whoami  pwd  systemname  bash$ **bash < tellme |& tee log**  Your current login and working directory are...  strike  /home/strike  bash: systemname: command not found  bash$ **ls**  tellme log  bash$ **cat log**  Your current login and working directory are  strike  /home/strike  bash: systemname: command not found |

Although UNIX shells are highly specialized and generally used interactively using the keyboard, a shell such as bash can also read input from a file. (After all, stdin is just a file.) In the previous snippet, the phrase bash < commands makes bash execute a list of commands found in the file tellme. The phrase |&tee log pipes the stdout and stderr of bash to the tee utility, which prints its stdin to stdout *and* to the file log.

But what if you want bash to process more than one input file? cat file1 file2 file3 | bash is a workable solution, and perhaps the only one, because bash doesn't support syntax like bash < file1 < file2 < file3.

Moreover, bash can't redirect output to more than one destination. For example, you can't enter an instruction like bighairyscript > ~/log | mail -s "Important stuff" team from the bash command line.

But a relatively new shell, the Z shell (zsh; see [Resources](http://www.ibm.com/developerworks/aix/library/au-unix-commandline/index.html#resources)), can process multiple input and output redirections within the same command line. For example, here's a command that saves stdout in a file called log and sends it to you using e-mail:

**Listing 6. Z shell**

|  |
| --- |
| zsh% **bash < tellme > log | mail -s "Who you are" 'whoami'**  bash: line 4: systemname: command not found  zsh% **<log**  Your current login, working directory, and system are...  strike  /home/strike |

(The phrase 'whoami' runs the command whoami and inserts the result of that command in place of the phrase. It's like running a little shell command before the rest of the command line runs.)

Let's walk through the previous command from left to right. The bash command creates the file log and mails the stdout of the commands found in tellme to you. Because stderr wasn't redirected by the > or the pipe, error messages are printed to stdout. The command <log is another Z shell shortcut; it's the same as cat. (And yes, the command > file is equivalent to cat > file.)

The Z shell can also process more than one input redirection. The Z shell command line cat < file1 < file2 < file3 is the same as cat file1 file2 file3. Admittedly, the former syntax is more unwieldy than the latter and, in general, multiple stdout redirection is used far more often. However, if the utility you want to run doesn't accept multiple input arguments, the Z shell's multiple input redirection can come in handy.

The Z shell is full of other novel tricks, including better *globbing* (wildcard matches), advanced pattern matches, and an extensive automatic completion system that minimizes what you have to type at the command line. The next two articles in this series will delve further into the Z shell.

Shell tricks

Here are some powerful command-line combinations that are sure to make you more productive. The commands should work in all shells, not just zsh.

* Create a verbatim copy of any directory, including symbolic links, with tar:

|  |
| --- |
| tar cf - */path/to/original* | \  (mkdir -p */path/to/copy*; cd */path/to/copy*; tar xvf -) |

The first tar archives the directory /path/to/original and emits the archive file to stdout; the hyphen (-) used with the create (c) option specifies stdout. The command in parentheses is a subshell: Commands in the subshell don't affect the environment of the current shell. mkdir -p creates the named directory, including any intermediate directories that need to be created; and cd changes to the new directory. The second tar reads an archive from stdin and expands it in place; the hyphen used with the extract (x) option refers to stdin.

* To save the stdout of a command sequence and view it at the same time, use less -O *file* . The -O option copies stdin to the named *file* . Here's an example:

|  |
| --- |
| sort /etc/aliases | less -Osorted |

* If a directory contains thousands of files, your shell (including zsh, depending on the number of files and their names) might not be able to enumerate all the files using wildcard matches, because the command line is typically limited to a certain number of characters. Hence, shell script phrases such as

|  |
| --- |
| foreach i (\*)  ...  end |

might fail. (You'll probably see a message like Line length exceeded when you exceed the length of your command line.) If such an error occurs, use a pipe and the xargs utility. xargs reads data from a pipe and runs a specified command for every line read.

For instance, if you want to find all Web pages on your server that reference www.example.com, you can use this command line:

|  |
| --- |
| % **find / -name '\*html' -print \**  **| xargs grep -l 'www.example.com' \**  **| less -Opages** |

xargs consumes the filenames from find and runs grep -l repeatedly to process every file, no matter how many files are named. (grep -l prints the name of the file if a match is found and then stops further matching in that file.) less allows you to page through the results and saves the list in the file named pages. The result is a list of filenames that contain the string "www.example.com".

Give your fingers a break, don't break your fingers

As [Part 1](http://www.ibm.com/developerworks/aix/library/au-unix-commandline/index.html) showed, the power of the UNIX command line is unmatched. With just a few keystrokes and a bit of syntactic *glue*, including pipes (|), tee, and redirection, you can assemble your own impromptu data transforms at each shell prompt.

For example, this command finds all of the text documents in your home directory that contain the words *Monthly Report*:

|  |
| --- |
| $ **find /home/joe -type f -name '\*.txt' -print | xargs grep -l "Monthly Report"** |

The command searches your entire home directory (find /home/joe) to find all regular files (-type f) with the suffix .txt, and then runs the grep command to search for the string Monthly Report. The -l option prints the file's name if a match was found. Hence, the output of the command is a list of files that match.

While the command above is useful, it's onerous to remember and retype, especially if you use the command regularly. Moreover, when the command line is your primary interface to e-mail, files, tools (such as editors, compilers, monitors), and remote systems, any time and effort you can save at the command line can be better spent on the task at hand. After all, a thousand few fractions of a second really add up.

To make light of repetitive tasks, UNIX shells provide a variety of helpful shortcuts, including:

* Sigils
* Wildcards
* A command history
* Environment variables
* Aliases
* Startup files

For example, you can refer to your home directory with the sigil ~ (tilde). You can also refer to your home directory using the $HOME environment variable, as shown in [Listing 1](http://www.ibm.com/developerworks/aix/library/au-speakingunix2.html#list1).   
**Listing 1. UNIX shell shortcuts**

|  |
| --- |
| $ **whoami**  strike  $ **echo ~**  /Users/strike  $ **echo $HOME**  /Users/strike  $ **!!**  echo $HOME  /Users/strike |

That last command, !! (two exclamation marks), might look a little strange, but it's a command history sigil that repeats the previous command verbatim. (Many shells also allow you to browse the list of previous commands using the up arrow key, or by pressing Control+P.)

Let's look at each kind of shell shortcut in more detail. This article is based on the Z shell (zsh -- see [Resources](http://www.ibm.com/developerworks/aix/library/au-speakingunix2.html#resources)), which is typically installed in /bin/zsh. (If your system doesn't have the Z shell, ask your system administrator to install it.) The Z shell has a few special features; otherwise, all the examples shown here work in all modern UNIX shells.

Shell sigils

Many command-line arguments are used so frequently that shells provide *sigils,* or symbols, as shorthand. You simply type the sigil in place of the argument.

As mentioned above, ~ refers to your home directory. A similar shorthand, ~*username* , refers to username's home directory. For example, ~joe refers to joe's home directory. So, to copy a file from joe's doc directory to your info directory, you could type:

|  |
| --- |
| $ **cp ~joe/doc/report.txt ~/info** |

Assuming that joe's home directory is in /guests and your home directory is /staff/bobr, ~joe is replaced with */guests/joe* and ~ becomes */staff/bobr,* finally yielding the command cp /guests/joe/doc/report.txt /staff/bobr/info. (See the sidebar, "[Proofing your work](http://www.ibm.com/developerworks/aix/library/au-speakingunix2.html#sidebar)" to learn how to preview your command line.)

**Proofing your work**

If you want to see what a command-line sigil expands to, use the echo command:

|  |
| --- |
| $ **echo ~joe/doc/report.txt ~/info**  /guests/joe/doc/report.txt /staff/bobr/info  $ **echo $SHELL**  /bin/zsh  $ **ls**  architecture.txt Services.pdf  services.txt Schema.pdf  $ **echo \*.txt**  architecture.txt services.txt |

The echo command emits whatever you type on the command line. However, because the shell expands (most) command-line arguments before invoking any program, the command prints the results of all substitutions. (The shell environment variable, $SHELL, contains the name of the currently running shell.)

Another valuable sigil is .. (two periods), shorthand for the directory immediately above the current directory. With .. and ., the sigil for the current working directory, you can refer to files and directories in the file system relative to your current working directory.

For instance, if your current working directory is ~/jane/projects/lambda, the shorthand ../.. refers to *the directory two directories above*, or ~/jane. To refer to the directory that contains ~/jane, you can use ../../../ ("three directories above") or the path ~jane/../. The latter path says *start at ~jane, and then go up one directory*.

To copy a file to your current directory, you need not name it; simply refer to it as . ("dot"):

|  |
| --- |
| $ **cp -pr /path/to/lots/of/stuff .** |

The former command recursively copies the /path/to/lots/of/stuff directory to your current directory, preserving the original time and date stamps. Path names that refer to .. and . are called *relative path names.* Path names that begin with a / (forward slash) or a ~ (tilde) are called *absolute path names* because you're referring to the file from the top of the file system, or from the top of a directory hierarchy.

Wildcards and patterns

With sigils, you reduce your typing time and can refer to a specific directory quickly and concisely. *Wildcards* are another form of shorthand to refer to the *contents of a directory*.

For example, assume that you have a directory containing 100 files. Some are C source code files that end with the suffix *.c*, others are object files with suffix *.o*, and still others are text documents (*.txt*), scripts (*.sh*), and executables (files with execute permission). To list only the C files, simply type:

|  |
| --- |
| $ **ls \*.c** |

The wildcard \* (typically called *star* rather than *asterisk*) means *match any sequence of characters*. The .c file name extension is a literal pattern that matches only a period followed by a lowercase *c.* So, \*.c means *any sequence of characters followed by a period and a lowercase c*. Given \*.c, the shell looks in the current directory (unless you provide a leading absolute or relative path name), finds every file name that matches the pattern, expands \*.c to that list of names, and passes the list as arguments to the ls command.

[Listing 2](http://www.ibm.com/developerworks/aix/library/au-speakingunix2.html#list2) demonstrates the use of \*.c based on the source code to wget, the command-line download utility.

**Z shell globs**

The Z shell has several unique and marvelous glob operators. Here are a few that stand out.

The \*\*/ glob operator expands to all directories below and includes the current working directory. Think of \*\*/ as a built-in find command. Referring to the wget source code again, you can find all the Makefiles with the command:

|  |
| --- |
| $ **echo \*\*/Makefile**  Makefile doc/Makefile po/Makefile  src/Makefile util/Makefile  windows/Makefile |

If you don't want to include the current working directory, simply type \*/\*\*/, as shown in this example:

|  |
| --- |
| $ **echo \*/\*\*/Makefile**  doc/Makefile po/Makefile  src/Makefile util/Makefile  windows/Makefile |

Another useful Z shell glob operator matches file *types.* Provide a pattern as before, but append (.), (/), (\*), or (@) to match regular files, directories, executable files, and symbolic links, respectively:

|  |
| --- |
| $ ls -d -F \*(/)  **ChangeLog-branches/ doc/ po/ src/ util/ windows/** |

Z shell also provides a shortcut for the (/) expression. Simply end a pattern with a forward slash to limit the results to directories:

|  |
| --- |
| $ **ls -d \*/**  ChangeLog-branches/ doc/ po/ src/ util/ windows/ |

**Listing 2. Use wildcards to find C source code files in a directory**

|  |
| --- |
| $ **ls \*.c**  alloca.c  ansi2knr.c  cmpt.c  connect.c  convert.c  ... |

The process of expanding a wildcard to the list of matching file names is called *globbing,* and UNIX shells have a variety of globbing operators (so-called *globs*) to help you express what you're looking for:

* The glob \* (star) matches any character or sequence of characters, including an empty sequence.
* The glob ? (question mark) matches any single character.
* The glob [ ] (square brackets) matches any of the enclosed characters. Within the brackets, you can refer to a range of characters by using - (hyphen), as in [a-z] or all lowercase letters.

(The Z shell has many unique glob operators. See the sidebar, [Z shell globs](http://www.ibm.com/developerworks/aix/library/au-speakingunix2.html#sidebar2) for more information.)

You can also repeat glob operators as necessary. [Listing 3](http://www.ibm.com/developerworks/aix/library/au-speakingunix2.html#list3) provides additional examples.

**Listing 3. Wildcard examples**

|  |
| --- |
| 1 $ **ls -1 -a -F**  ./libs  ChangeLog  ChangeLog-branches/  Makefile  Makefile.in  alloca.c  ansi2knr.c  cmpt.c  cmpt.o  config.h  config.h.in  connect.c  connect.h  connect.o  convert.c  convert.h  convert.o  ...  wget\*  2 $ **ls -a -F .\***  ./lib  3 $ ls -1 \*.?  alloca.c  ansi2knr.c  cmpt.c  cmpt.o  config.h  connect.c  connect.h  connect.o  convert.c  convert.h  convert.o  ...  4 $ **ls -1 ????.?**  cmpt.c  cmpt.o  5 $ **ls [a-c]?\*.\***  alloca.c  ansi2knr.c  cmpt.c  cmpt.o  config.h  config.h.in  connect.c  connect.h  connect.o  convert.c  convert.h  convert.o  cookies.c  cookies.h  cookies.o |

In [Listing 3](http://www.ibm.com/developerworks/aix/library/au-speakingunix2.html#list3), Command 1 shows all the entries in the directory, including those entries that begin with . (dot) in a long list. (The -a option shows the so-called *dot files*; the -1 option lists everything in one column; and the -F option highlights directories with a / (forward slash) and executables with a \* (star).)

Command 2 finds each entry whose name begins with a dot (hence .\*). The third command finds only those items that have a one-letter suffix.

The fourth command finds only those items that have four characters followed by a dot and one character. Finally, Command 5 finds items that begin with lowercase *a,* lowercase *b,* or lowercase *c* and are followed by at least one letter, then anything, then a period, and then any suffix. As you can see, you can repeat the glob operators as needed.

So, what would ls \*.z yield (assuming no such files exist)? It yields a helpful error message:

|  |
| --- |
| $ **ls \*.z**  zsh: no matches found: \*.z |

A bit of (command) history

So far, you've seen how to specify paths and pick and choose files. You can express yourself at the command line. However, even if all command lines were short and sweet, chances are you would still get tired of typing the same thing over and over again. In particular, you would probably get weary of typing long, complex command lines with loads of options, or where the order of the arguments has to be just so. Luckily, most shells maintain a *history* of previous commands. To rerun a command, you simply find its entry in the history list and rerun it. And like other parts of the shell, shortcuts make references quick and easy.

To enable command history in Z shell, type:

|  |
| --- |
| $ **HISTSIZE=500**  $ **SAVEHIST=500** |

Here, the commands specify that both the shell and the persisted history file should retain the last 500 commands. (By default, Z shell saves only the last 30 commands.) Check your shell's documentation for information on how to capture and persist command histories.

After working in the shell a while, you can view your command history by simply typing history:

|  |
| --- |
| $ **history**  ...  781 /bin/ls -d \*/  782 /bin/ls -F \*(/)  783 /bin/ls -d -F \*(/)  784 /bin/ls -d -F \*/  785 /bin/ls -d \*/ |

Each command you run is assigned a sequential, numerical identifier. You use that identifier, such as *782,* to refer to an entire command and to parts of each command. To rerun a command verbatim, type ! (exclamation mark) followed by the command's number:

|  |
| --- |
| $ **!785**  ChangeLog-branches/ doc/ po/ src/ util/ windows/ |

If you want a specific argument from a historical command, refer to the command with a ! (exclamation mark) and provide *:N,* where **0** refers to the command name, **1** refers to the first argument, and so on. For example, to extract the second argument of command 782 in the history log, type the code shown in [Listing 4](http://www.ibm.com/developerworks/aix/library/au-speakingunix2.html#list4).

**Listing 4. Extract the second argument from command 782**

|  |
| --- |
| $ **echo !782:2**  echo \*(/)  ChangeLog-branches doc po src util windows  $ ls AUTHORS COPYING INSTALL MACHINES  AUTHORS COPYING INSTALL MACHINES  $ **echo !!:3**  echo INSTALL  $ **history -2**  788 ls AUTHORS COPYING INSTALL MACHINES  789 echo INSTALL  $ **echo !788^**  echo AUTHORS  AUTHORS  $ **echo !788$**  echo MACHINES  MACHINES |

The command history -2 prints the previous two commands. As shortcuts, you can refer to the first argument of a command (not the command name itself) using ^ (carat), and you can refer to the last argument of a historical command with the shortcut $ (dollar sign). You can also refer to a range of arguments using a range notation, as shown in [Listing 5](http://www.ibm.com/developerworks/aix/library/au-speakingunix2.html#list5).

**Listing 5. A range notation**

|  |
| --- |
| $ **echo AUTHORS COPYING INSTALL MACHINES**  AUTHORS COPYING INSTALL MACHINES  $ **echo !!:1-2**  echo AUTHORS COPYING  AUTHORS COPYING |

There are also other, more direct ways to recall historical commands. One way is to search for it:

|  |
| --- |
| $ **ls I\***  $ **ls M\***  $ **echo !?M**  ls INSTALL |

The construct !?M asks for the most recent historical command line that contains an uppercase letter *M.*

Environment variables

Speaking fluent *command line* is an essential UNIX skill. But speaking UNIX doesn't stop at the shell prompt -- you must also communicate with the myriad of UNIX utilities. In UNIX, environment variables retain settings in your shell and allow you to propagate your preferences to each and every utility you launch from the command line.

Some environment variables -- called *shell variables* -- are used only by your shell to control its behavior. For instance, only the Z shell uses $HISTSIZE and $SAVEHIST, shown above, to manage command histories. Think of shell variables as settings.

Other environment variables are *exported*, or made globally available, and are copied into the process space (the *environment*) of every command you launch from the command line. For example, $HOME is a special environment variable that retains the location of your home directory. The UNIX login sequence sets $HOME (and other environment variables), and then starts your shell, which in turn uses $HOME to find all your shell startup files. Other applications that you launch, such as SSH and FTP, refer to $HOME to find your .netrc file (used to store confidential, remote access passwords). Some environment variables -- such as $HOME, $PATH, and $SHELL -- every application uses. Other environment variables might be unique to an application.

To see all your current environment variables, type printenv, as shown in [Listing 6](http://www.ibm.com/developerworks/aix/library/au-speakingunix2.html#list6). (Depending on how your system administrator configured your system, you might have many more, or far fewer, environment variables than are shown here.)

**Listing 6. View environment variables**

|  |
| --- |
| $ **printenv**  PATH=/Users/strike/bin:/Applications/xampp/xamppfiles/bin:/Users/strike/bin:/usr/bin:/  bin:/usr/sbin:/sbin  HOME=/Users/strike  SHELL=/bin/zsh  USER=strike  TERM=xterm-color  LOGNAME=strike  SHLVL=1  PWD=/Local/src/versions/wget/wget-1.9  OLDPWD=/Local/src/versions/wget/wget-1.9/src  PERL5LIB=/Applications/xampp/xamppfiles/lib/perl5/site\_perl/5.8.7:/Projects/IGSP/src  CLICOLOR=true  MANPATH=/Local/root/share/man:/usr/share/man:/opt/local/share/man  INFOPATH=/opt/local/share/info  LESS=-n |

You likely recognize many of these variables; others might be new. The shell level ($SHLVL) shows how many shells deep you are. A **1** indicates a login shell; a **2** means that you launched another shell from your login shell, and so on. You can use the value of $SHLVL to change your prompt for each subsequent, nested shell. $TERM reflects your terminal (probably terminal emulator) settings -- important for ensuring proper rendering of text, colors, as well as proper interpretation of keystrokes. $PWD is your current working directory, while $OLDPWD is your previous working directory. You can use both variables to quickly go back and forth between two directories, as shown in [Listing 7](http://www.ibm.com/developerworks/aix/library/au-speakingunix2.html#list7).

**Listing 7. Toggle between directories**

|  |
| --- |
| $ **echo $PWD**  /Users/strike  $ **echo $OLDPWD**  /Local/src/versions/wget/wget-1.9  $ **cd $OLDPWD**  $ **echo $PWD**  /Local/src/versions/wget/wget-1.9  $ **echo $OLDPWD**  /Users/strike |

The remaining environment variables in the list above are application-specific. Each retains preferences that control how each associated application works when you launch it. $PERL5LIB is a search path for Perl to find custom libraries. The ls command uses $CLICOLOR to render file types in color (directories in blue, executables in green, and so on). Custom application environment variables are typically documented in the program's man pages.

Setting an environment variable is identical to setting a shell variable. However, you must export the variable to make it globally available:

|  |
| --- |
| $ **MYVARIABLE=$HOME/projectX**  $ **export TMPDIR=/tmp/projectX** |

The former command sets a shell variable named $MYVARIABLE. (The leading dollar sign is the shell prompt. When you set a variable, you don't provide the $. However, you do need the dollar sign, as in $MYVARIABLE, whenever you use the variable.) $MYVARIABLE is visible only to the shell, because it wasn't exported. To see a list of all shell variables, type set. The output of set includes the environment variables, because those are available to the shell as well.

In the latter command, $TMPDIR is set, exported, and available to all applications launched from the shell. One application that uses $TMPDIR is the GNU Compiler Collection (GCC) compiler. The value you store in $TMPDIR is where GCC generates its temporary files.

If you want to remove an environment variable, simply type unset and the name of the variables, as shown in [Listing 8](http://www.ibm.com/developerworks/aix/library/au-speakingunix2.html#list8).

**Listing 8. Remove an environment variable**

|  |
| --- |
| $ **set**  HOME=/Users/strike  MYVARIABLE=/Users/strike/projectX  TMPDIR=/tmp/projectX  ...  $ **unset MYVARIABLE TMPDIR**  $ **set**  HOME=/Users/strike  .... |

Aliases and startup files

The previous sections might have you concerned about just how much you have to type at the command line. Yes, there's a lot to learn -- this is because the shell environment is so rich. Remember, though, that with great power comes great productivity (many apologies to Spider-man).

To conserve those precious keystrokes and retain all the settings you've made, UNIX shells offer aliases and startup files, respectively. *Aliases* are shortcuts that you create. *Startup files* are read each time your shell starts and are the ideal place to store (and share) all your shell settings, such as shell variables (options), environment variables, and aliases.

An alias is a short sequence that you use instead of a longer command. You can think of an alias as a nickname for a command line. Instead of typing:

|  |
| --- |
| $ find /home/joe -type f -name '\*.txt' -print | xargs grep -l "Monthly Report" |

at the command prompt, you might type a nickname that you created:

|  |
| --- |
| $ **findreports** |

The shell does the heavy lifting, replacing findreports with its expansion. To create the findreports alias, type:

|  |
| --- |
| alias findreports='find $HOME -type f -name "\*.txt" -print |  xargs grep -l "Monthly Report"' |

Single quotation marks must delimit each alias. If you need quotation marks inside the alias, use double quotation marks. Z shell aliases can contain many shell primitives, including variables, pipes, redirection, other aliases, and other shell operands, as shown in [Listing 9](http://www.ibm.com/developerworks/aix/library/au-speakingunix2.html#list9).

**Listing 9. Z shell primitives**

|  |
| --- |
| $ **alias ll='/bin/ls -l'**  $ **ll -d 2002\***  drwxrwxr-x 2 www-data www-data 4096 Jan 16 2002 2002-02  drwxrwxr-x 2 www-data www-data 4096 Jan 22 2002 2002-03  drwxrwxr-x 2 www-data www-data 4096 Apr 15 2002 2002-04  drwxrwxr-x 2 www-data www-data 4096 Apr 19 2002 2002-05  ...  $ **alias lt='ll -t'**  $ **lt -d 2002\***  drwxrwxr-x 2 www-data www-data 4096 Apr 19 2002 2002-05  drwxrwxr-x 2 www-data www-data 4096 Apr 15 2002 2002-04  drwxrwxr-x 2 www-data www-data 4096 Jan 22 2002 2002-03  drwxrwxr-x 2 www-data www-data 4096 Jan 16 2002 2002-02  $ **alias m='pinky | grep mstreicher'**  $ **m**  mstreicher Martin Streicher ...  $ **alias snap='pinky >> ~/.pinky'**  $ **snap**  $ **snap**  $ **cat ~/.pinky**  Login Name TTY Idle When Where  mstreicher Martin Streicher pts/0 Jun 18 16:40 cpe-071-065-224-025.nc.res.rr.com  Login Name TTY Idle When Where  mstreicher Martin Streicher pts/0 Jun 18 16:40 cpe-071-065-224-025.nc.res.rr.com |

The alias ll refers to /bin/ls -- absolute paths are never replaced by alias substitution.

When you type ll, it's replaced by its alias, and any remaining command-line arguments are appended. Hence, ll -d 2002\* is really the command /bin/ls -l -d 2002\*. The alias lt refers to ll and adds the -t flag to sort by creation time. The lt alias expands to /bin/ls -l -t -d 2002\*. The m alias includes a pipe. The snap alias uses redirection to append the output of a command to a file.

To see all the aliases set in your shell, just type alias (with no arguments), as shown in [Listing 10](http://www.ibm.com/developerworks/aix/library/au-speakingunix2.html#list10).

**Listing 10. View all the aliases in your shell**

|  |
| --- |
| $ **alias**  alias findreports='find $HOME -type f -name "\*.txt" -print | xargs grep -l  "Monthly Report"'  alias ll='/bin/ls -l'  alias lt='ll -t'  alias m='pinky | grep mstreicher'  alias snap='pinky >> ~/.pinky'  ... |

If you want to remove an alias, just type unalias and the alias's name. You can also list multiple aliases at a time, as shown in [Listing 11](http://www.ibm.com/developerworks/aix/library/au-speakingunix2.html#list11).

**Listing 11. View multiple aliases simultaneously**

|  |
| --- |
| $ **unalias m snap**  $ **alias**  alias findreports='find $HOME -type f -name "\*.txt" -print | xargs grep -l  "Monthly Report"'  alias ll='/bin/ls -l'  alias lt='ll -t' |

Finally, after you've worked hard to set up your environment just so, you'll want to keep your settings for next time. Indeed, you want your shell to be consistent from session to session and from instance to instance -- say, when multiple terminal windows are open on your workstation.

Shells include startup files to (re)initialize your environment when your shell starts. Startup files can be simple -- just a list of variables and values -- or quite complex, including customization logic and elaborate functions. Some users keep many sets of startup files, one set per project.

Z shell uses the startup files .zshrc and .zprofile, both of which reside in your home directory. (Other shells have similar files with similar names, and you can read your shell documentation for specifics. Some shells also provide for *shutdown* files, or files to run when your shell is exiting.) The .zshrc file is *sourced*, or read, and processed whenever you start a new shell; the .zprofile file is sourced only when you start a login shell.

After you've configured your shell, take a snapshot of your settings and save them in one of the shell startup files:

|  |
| --- |
| $ **set >> $HOME/.zshrc**  $ **alias >> $HOME/.zshrc** |

**Note:** You might want to edit the resulting .zshrc file and remove variables that are session-specific.

More power

Whew! This installment of *Speaking UNIX* covered a lot of ground, but your diligence should yield vast rewards. Work smarter, not harder, and save the extra time to do really important things, like play slashem.

Next time, *Speaking UNIX* goes positively old school. I'll forgo those trendy browsers and examine how to connect, download, upload, transfer, and communicate entirely from the command line.

Stay tuned.