

Sistemas Operativos

Shell Scripts

Ricardo Ruiz Rodríguez

Instituto de Computación
Universidad Tecnológica de la Mixteca
Primavera 2014

Índice I

1 Manipulating file attributes

- Introduction
- File types
- Permissions

2 Processes

- Introduction
- Foreground and background processes
- Dealing with processes

3 Variables

- Overview
- Array variables
- Further considerations
- Types of variables

4 Substitution

Índice II

- Filename substitution (globbing)
- Value-based variable substitution
- Command and arithmetic substitution

5 Quoting

- Quoting schemes
- Quoting rules and situations

Introduction

In addition to working with files and directories, shell scripts are often called on to manipulate the attributes of a file.

Now, you will learn how to manipulate the following file attributes:

- ① Permissions
- ② Owners
- ③ Groups

To determine a file's type, specify the **-l** option to the **ls**. When this option is specified, **ls** lists the file type for the specified files. To obtain file type information about a directory, you must specify the **-d** option along with the **-l** option

File types

UNIX supports several different types of files.

Files can contain your important data, such as files from a word processor or graphics package, or they can represent devices, directories, or symbolic links.

The special characters for different file types are:

- ❶ -: Regular file.
- ❷ l: Symbolic link.
- ❸ c: Character special.
- ❹ b: Block special.
- ❺ p: Named pipe.
- ❻ s: Socket.
- ❼ d: Directory file.

Regular files

- Regular files are the most common type of files you will encounter. These files store any kind of data.
- Often simply determining that a file is a regular file tells you very little about the file itself.
- Usually you need to know whether a particular file is a binary program, a shell script, or a library. In these instances, the **file** command is very useful.

Symbolic links

- A symbolic link is a special file that points to another file on the system. When you access one of these files, it has a pathname stored inside it.
- A symbolic link is similar to a shortcut or an alias in Windows or Mac OS.
- You can use symbolic links to make a file appear as though it is located in many different places or has many different names in the file system.
- Symbolic links can point to any type of file or directory.

Create symbolic links using the **ln** command with the **-s** option.

The syntax is as follows:

ln -s *source destination*

Here, *source* is either the absolute or relative path to the original version of the file, and *destination* is the name you want the link to

Device files

- You can access UNIX devices through reading and writing to device files. These device files are access points to the device within the file systems.
- Usually, device files are located under the **/dev** directory. The two main types of device files are:
 - ① Character special files.
 - ② Block special files.
- **Character special files** provide a mechanism for communicating with a device one character at a time.
- **Block special files** also provide a mechanism for communicating with device drivers via the file system. These files are called block devices because they transfer large blocks of data at a time.

Named pipes and sockets

- On the command line, temporary anonymous pipes are used, but sometimes more control is needed than the command line provides.
- UNIX provides a way to create a named pipe, so that two or more process can communicate with each other via a file that acts like a pipe.
- Because these files allow process to communicate with one another, they are one of the most popular forms of IPC.
- Socket files are another form of IPC, but sockets can pass data and information between two processes that are not running on the same machine.
- Socket files are created when communication to a process on another machine located on a network is required.

Owners, groups and permissions

Every file in UNIX has the following attributes:

- 1 Owner permissions.
- 2 Group permissions.
- 3 Other (world) permissions.

You can perform the following actions on a file:

- 1 Read (r).
- 2 Write (w).
- 3 Execute (x).

If a user has *read* permissions, that person can view the contents of a file. A user with *write* permissions can change the contents of a file, whereas a user with *execute* permissions can run a file as a program.

Directory permissions I

- The **x** bit on a directory grants access to the directory. The read and write permissions have no effect if the access bit is not set.
- The read permission on a directory enables users to use the **ls** command to view files and their attributes that are located in the directory.
- The write permission on a directory is the permission to watch out for because it lets a user add and also remove files from the directory.
- A directory that grants a user only execute permission will not enable the user to view the contents of the directory or add or delete any files from the directory, but it will let the user run executable files located in the directory.

Directory permissions II

- To ensure that your files are secure, check both the file permissions and the permissions of the directory where the file is located.
- If a file has write permission for owner, group, and other, the file is insecure.
- If a file is in a directory that has write and execute permissions for owner, group, and other, all files located in the directory are insecure, no matter what the permissions on the files themselves are.

SUID and GUID file permission I

- Often when a command is executed, it will have to be executed with special privileges in order to accomplish its task.
- When you change your password with the **passwd** (/usr/bin/passwd) command, your new password is stored in the file **/etc/shadow**. As a regular user, you do not have read or write access to this file for security reasons, but when you change your password, you need to have write permission to this file.
- Additional permissions are given to programs via a mechanism known as the Set User ID (SUID) and Set Group ID (SGID) bits.

SUID and GUID file permission II

- When you execute a program that has the SUID bit enabled, you inherit the permissions of that program's owner.
- Programs that do not have the SUID bit set are run with the permissions of the user who started the program.
- This is true for SGID as well.
- The SUID and SGID bits will appear as the letter "s" if the permission is available.
- The SUID "s" bit will be located in the permission bits where the owners execute permission would normally reside.
- A capital letter "S" in the execute position instead of a lowercase s indicates that the execute bit is not set.

SUID and GUID file permission III

- The SUID bit or **stick bit** imposes extra file removal permissions on a directory. A directory with write permissions enabled for a user enables that user to add and delete any files from this directory.
- Directories can also take advantage of the SGID bit.
- If a directory has the SGID bit set, any new files added to the directory automatically inherit that directory's group, instead of the group of the user writing the file.

Changing permissions I

You can change file and directory permissions with the **chmod** command. The basic syntax is as follows:

chmod *expression files*

Here, *expression* is a statement of how to change the permissions. This expression can be of the following types:

- Symbolic.
- Octal.

The symbolic expression has the syntax of:

(*who*)(*action*)(*permissions*)

Here, *who*:

- **u** Owner.
- **g** Group.

Changing permissions II

- **o** Other.
- **a** All.

action:

- **+** Adding permissions to the file.
- **-** Removing permission from the file.
- **=** Explicitly set the file permissions.

permissions:

- **r** Read.
- **w** Write.
- **x** Execute.
- **s** SUID or SGID.

Changing permissions III

What does each command?

```
user@machine$ chmod a=r *
user@machine$ chmod guo=r *
user@machine$ chmod go-w dummy_file
user@machine$ cd ; chmod go= *
user@machine$ cd ; chmod go-rwx *
user@machine$ chmod guo+rx *
user@machine$ chmod uog+xr *
user@machine$ chmod go-w,a+x a.out
user@machine$ cd ; chmod ug+s .
```

- **chmod** also enables you to change the permissions for every file in a directory including the files in subdirectories.

Changing permissions IV

- You can accomplish this by specifying the **-R** option.
- By changing permissions with an octal expression, you can explicitly set file permissions:

r w x

0 0 0 = 0

0 0 1 = 1

0 1 0 = 2

0 1 1 = 3

1 0 0 = 4

1 0 1 = 5

1 1 0 = 6

1 1 1 = 7

Changing owners and groups

The **chown** command changes the ownership of a file.

The basic syntax is as follows:

chown *options user:group files*

Because considerable variation exists in the available *options*, please consult the *man* page on your system for a complete list.

- The value of *user* can be either the name of a user or the user id (uid) of a user on the system.
- The value of *group* can be the name of a group or the group id (gid) of a group on the system.
- To just change the owner, you can omit the group value.

Introduction

- In UNIX every program runs as a process.
- Whenever you issue a command in UNIX, it creates, or starts, a new process.
- The operating system tracks processes through a number known as the pid or process ID.
- Each process in the system has a *unique* pid.
- Pids eventually repeat because all the possible numbers are used up and the next pid rolls or starts over.
- At any one time, no two processes with the same pid exist in the system.
- When you start a process, there are two ways you can run it: in the foreground or background.
- The difference is how the process interacts with you at the terminal.

Foreground and background processes I

- By default, every process that you start runs in the **foreground**.
- It gets its input from the keyboard and sends its output to the screen.
- While one command is running, you can not run any other commands (start any other processes).
- UNIX provides facilities for starting processes in the background, suspending foreground processes, and moving processes between the foreground and background.
- A **background** process runs without being connected to your keyboard. If the background process requires any keyboard input, it waits.

Foreground and background processes II

- The advantage of running a process in the background is that you can run other commands.
- The simplest way to start a background process is to add an ampersand (&) at the end of the command.
- When you run a command in background, the first line you will see contains information about the background process: the job number and process ID.
- You need to know the job number to manipulate it between background and foreground.
- As homework, check the **set -o**, **set -o monitor** and **stty -a** commands.

Moving processes I

- In addition to running a process in the background using `&`, you can move a foreground process into the background.
- While a foreground process runs, the shell does not process any new commands. Before you can enter any commands, you have to *suspend* the foreground process to get a command prompt.
- When a foreground process is suspended, the original process is still in memory but is not getting any CPU time.
- To resume the foreground process, you have two choices: background and foreground.

Moving processes II

- The **bg** command enables you to resume the suspended process in the background; the **fg** command returns it to the foreground.
- By default, the **bg** command moves the most recently suspended process to the background.
- You can have multiple processes suspended at one time. To differentiate them, put the job number prefixed with a percent sign (%) on the command line.
- When you have a process that is in the background or suspended, you can move it to the foreground with the **fg** command.
- By default, the process most recently suspended or moved to the background moves to the foreground.

Moving processes III

- You can also specify which job, using its job number, you want to make foreground.

Keeping background processes around

- You can prevent a background process from terminating, which is the default action, when you sign off or are disconnected.
- The **nohup** command prevents your process from getting the HUP (Hang UP) signal and enables it to continue processing.
- The **nohup** command is simple to use: just add it before the command you actually want to run.
- Because **nohup** is designed to run when there is no terminal attached, it wants you to redirect output to a file. If you do not, **nohup** redirects it automatically to a file known as *nohup.out*.

Waiting for background processes to finish I

- There are two ways to wait for a background process to finish before doing something else:
 - ① You can press the Enter key every few minutes until you get the completion message.
 - ② You can use the **wait** command.
- There are three ways to use the wait command:
 - ① With no options (the default).
 - ② With a process ID.
 - ③ With a job number prefixed with a percent sign.
- The command will wait for the completion of the job or process you specify.
- If you do not specify a job or process (the default setting), the **wait** command waits for all background jobs to finish.

Waiting for background processes to finish II

- Using **wait** without any options is useful in a shell script that starts a series of background jobs. When they are all done, it can continue processing.

Listing running processes I

- You can start processes in the foreground and background, suspend them, and move them between the foreground and background, but how do you know what is running?
- There are two commands to help you find out: **jobs** and **ps**.
- The **jobs** command shows you the processes you have suspended and the ones running in the background.
- Another command that shows all processes running is the **ps** (process status) command.
- By default, it shows those processes that you are running.
- There are different flavors, or versions of UNIX. **ps** is one command where the differences are very obvious.

Listing running processes II

- Use the `man ps` command for an explanation of the states and options available.
- One of the most commonly used flags for **ps** is the **-f** (full) option, which provides *full* information.

Killing a process I

- Another handy command to use with jobs and processes is the **kill** command.
- As the name implies, the **kill** command kills, or ends, a process.
- Just like the **fg** and **bg** commands, the job number is prefixed with a percent sign.
- You can also kill a specific process by specifying the process ID on the command line without the percent sign used with job numbers.
- In reality, **kill** does not physically kill a process; it sends the process a signal.

Killing a process II

- By default, it sends the TERM (value 15) signal. A process can choose to ignore the TERM signal or use it to begin an orderly shut down (flushing buffers, closing files, and so on).
- If a process ignores a regular kill command, you can use **kill -9** or **kill -KILL** followed by the process ID or job number (prefixed with a percent sign). This forces the process to end.

Parent and child processes I

- Each process has two ID numbers assigned to it: process ID (PID) and parent process ID (PPID).
- When a child is forked or created, from its parent, it receives a copy of the parent's environment, including environment variables.
- The child can change its own environment, but those changes do not reflect in the parent and go away when the child exits.
- Background and suspended processes are usually manipulated via job number (job ID). In addition, a job can consist of multiple processes running in series or at the same time, in parallel, so using the job ID is easier than tracking the individual processes.

Parent and child processes II

- Whenever you run a shell script, in addition to any commands in the script, another copy of the shell interpreter is created. This new shell is known as a **subshell**.
- In addition to creating (forking) child processes, you can *overlay* the current process with another.
- The **exec** command replaces the current process with the new one.
- Use this command only with great caution.

Overview I

- Variables are "words" that hold a value.
- The shell enables you to create, assign, and delete variables.
- Although the shell manages some variables, it is mostly up to the programmer to manage variables in shell scripts.

Variables are defined as follows:

`name=value`

Here, *name* is the name of the variable, and *value* is the value it should hold.

For example:

`FRUIT=peach`

defines the variable *FRUIT* and assigns it the value *peach*.

- Variables of this type are called **scalar variables**.

Overview II

- A scalar variable can hold only one value at a time.
- Scalar variables are also referred to as *name value* pairs, because a variable's name and its value can be thought of as a pair.
- The name of a variable can contain only letters (a to z or A to Z), numbers (0 to 9) or the underscore character (_). In addition, a variable's name can start only with a letter or an underscore.
- Variable names, such as 1, 2 or 11, that start with numbers are reserved for use by the shell.
- You can use the value stored in these variables, but you can not set the value yourself.

Overview III

- The reason you cannot use other characters such as **!**, *****, or **-** is that these characters have a special meaning for the shell.
- If you try to make a variable name with one of these special characters it confuses the shell.
- The shell enables you to store any value you want in a variable.
- The one thing to be careful about is using values that have spaces.
- In order to use spaces you need to *quote* the value.
- To access the value stored in a variable, prefix its name with the pesos sign (**\$**).

Array variables I

- **Arrays** provide a method of grouping a set of variables.
- Instead of creating a new name for each variable that is required, you can use a single array variable that stores all the other variables.
- The simplest method of creating an array variable is to assign a value to one of its indices. This is expressed as follows:

`name[index]=value`

Here *name* is the name of the array, *index* is the index of the item in the array that you want to set, and *value* is the value you want to set for that item.

- As an example, the following commands:

Array variables II

```
$ FRUIT[0]=apple  
$ FRUIT[1]=banana  
$ FRUIT[2]=orange
```

set the values of the first three items in the array named FRUIT. You could do the same thing with scalar variables as follows:

```
$ FRUIT_0=apple  
$ FRUIT_1=banana  
$ FRUIT_2=orange
```

Although this works fine for small numbers of items, the array notation is much more efficient for large numbers of items.

Array variables III

- It is not necessary to set the array indices in sequence. For example, if you issue the command:

```
$ FRUIT[10]=cranberry
```

the value of the item at index 10 in the array FRUIT is set to cranberry.
- One thing to note here is that the shell does not create a bunch of blank array items to fill the space between index 2 and index 10. Instead, it keeps track of only those array indices that contain values.
- If an array variable with the same name as a scalar variable is defined, the value of the scalar variable becomes the value of the element of the array at index 0. For example, if the following commands are executed:

Array variables IV

```
$ FRUIT=apple  
$ FRUIT[1]=peach
```

the element `FRUIT` has the value `apple`. At this point any accesses to the scalar variable `FRUIT` are treated like an access to the array item `FRUIT[0]`.

- The second form of array initialization is used to set multiple elements at once. In *bash*, the multiple elements are set as follows:

```
name=(value1 ... valueN)
```

Here, *name* is the name of the array and *values1 ... valueN*, are the values of the items to be set.

Array variables V

- When setting multiple elements at once, the shell uses consecutive array indices beginning at 0.
- When setting multiple array elements in bash, you can place an array index before the value:
`myarray=([0]=derri [3]=gene [2]=mike [1]=terry)`

Accessing array values

- After you have set any array variable, you access it as follows:

`${name[index]}`

Here *name* is the name of the array, and *index* is the index that interests you.

- You can access all the items in an array in one of the following ways:

`${name[*]}`

`${name[@]}`

Here *name* is the name of the array you are interested in.

- If any of the array items hold values with spaces, the first form of array access will not work and will need to use the second form.
- The second form quotes all the array entries so that embedded spaces are preserved.

Read-only variables I

- The shell provides a way to mark variables as read-only by using the **readonly** command.
- After a variable is marked read-only, its value cannot be changed. Consider the following commands:

```
$ FRUIT=kiwi  
$ readonly FRUIT  
$ echo $FRUIT  
kiwi  
$ FRUIT=aguacate
```

The last command results in an error message.

Read-only variables II

- The **echo** command can read the value of the variable `FRUIT`, but the shell did not enable you to overwrite the value stored in the variable `FRUIT`.
- This feature is often used in scripts to make sure that critical variables are not overwritten accidentally.

Unsetting variables

- Unsetting a variable tells the shell to remove the variable from the list of variables that it tracks.
- This is like asking the shell to forget a piece of information because it is no longer required.
- Both scalar and array variables are unset using the unset command:

unset name

Here *name* is the name of the variable to unset.

- You cannot use the unset command to unset variables that are marked readonly.

Types of variables

When a shell is running, three main types of variables are present:

- ❶ **Local Variables:** is a variable that is present within the current instance of the shell. It is not available to programs that are started by the shell. The variables that you looked at previously have all been local variables.
- ❷ **Environment Variables:** is a variable that is available to any child process of the shell. Some programs need environment variables in order to function correctly. Usually a shell script defines only those environment variables that are needed by the programs that it runs.
- ❸ **Shell Variables:** is a special variable that is set by the shell and is required by the shell in order to function correctly. Some of these variables are environment variables whereas others are local variables.

Exporting environment variables I

- You place variables in the environment by *exporting* them.
- Exporting can be done as follows:

export name

- This command marks the variable with the specified *name* for export.
- This is the only form supported by *sh*, thus it is the most commonly encountered form.
- The standard shell idiom for exporting environment variables is:

name=value ; **export** name

Exporting environment variables II

- Usually the assignment statement of an environment variable and the corresponding export statement are written on one line to clarify that the variable is an environment variable.
- You can also use the export command to export more than one variable to the environment:

export name1 name2 ... nameN

- A second form for exporting variables is supported by *ksh* and *bash*:

export name=value

In this form, the variable specified by *name* is assigned the given *value*. Then that variable is marked for export.

Shell variables I

- The variables that you have examined so far have all been user variables.
- A user variable is one that the user can manually set and reset.
- Shell variables, are variables that the shell sets during initialization and uses internally.
- A partial list of shell variables available in *sh*, *ksh*, and *bash* are:

PWD	REPLY	IFS
UID	RANDOM	PATH
SHLVL	SECONDS	HOME

Introduction I

- The shell performs substitution when it encounters an expression that contains one or more special characters.
- You have learned already how to access a variable's value using the special \$ character.
- The process of retrieving the value of a variable is called **variable substitution**.
- In addition to this type of substitution, the shell can perform several other types of substitutions:
 - Filename substitution (called globbing).
 - Value-based variable substitution.
 - Command substitution.
 - Arithmetic substitution.

Filename substitution

- The most common type of substitution is filename substitution. It is sometimes referred to as *globbing*.
- Filename substitution is the process by which the shell expands a string containing wildcards into a list of filenames.
- Wildcards used in filename substitution are the following:
 - ❶ *: Matches zero or more occurrences of any character.
 - ❷ ?: Matches one occurrence of any character.
 - ❸ **[characters]**: Matches one occurrence of any of the given *characters*.

Using the * Wildcard I

- The simplest form of filename substitution is the ***** character. The ***** tells the shell to match zero or more occurrences of any character.
- If given by itself, it matches all filenames.
- Using the ***** character by itself is required in many cases, but its strength lies in the fact that you can use it to match file *suffixes*, *prefixes*, or *both*.
- To match a **file prefix**, use the ***** character as follows:
command prefix*

Here, *command* is the name of a command, such as **ls**, and *prefix* is the filename prefix you want to match.

Using the * Wildcard II

- To match a **file suffix**, you use the ***** character as follows:
command **suffix*
- Here, *command* is the name of a command, such as **ls**, and *suffix* is the filename suffix you want to match.
- You can match both the suffix and the prefix of files using the ***** character as follows:

command *prefix*suffix*

Here, *command* is the name of a command, such as **ls**, *prefix* is the filename prefix, and *suffix* is the filename suffix you want to match.

- You can also use more than one occurrence of the ***** character to narrow down the matches.

Using the * Wildcard III

- For example, if the command

```
$ ls CGI*.java
```

matches the following files:

```
CGI.java CGIGet.java CGIPost.java CGITester.java
```

and you want to list only the files that start with the characters *CGI*, end with *.java*, and contain the characters *st*, you can use the following command:

```
$ ls CGI*st*.java
```

The output is:

```
CGIPost.java CGITester.java
```

- Globbing is **case sensitive**.

Using the ? Wildcard

- One limitation of the * wildcard is that it matches one or more characters each time.
- Consider a situation where you need to list all files that have names of the form ch0X.tex, where X is a single number or letter.

```
$ ls ch0*.tex
```

- In order to match only one character, the shell provides you with the ? wildcard

```
$ ls ch0?.tex
```

- Say that you now want to look for all files that have names of the form chXY, where X and Y are any number or character. To accomplish this you can use the command:

```
$ ls ch??.*
```

Matching sets of characters I

- Two potential problems with the `?` and `*` wildcards are:
 - ① They match any character, including special characters such as hyphens (`-`) or underlines (`_`).
 - ② You have no way to indicate that you want to match only letters or only numbers to these operators.
- Sometimes you need more control over the exact characters that you match.
- Consider the situation where you want to match filenames of the form `ch0X`, where `X` is a number between 0 and 9.
- Neither the `*` or the `?` operator is cut out for this job.
- Fortunately, the shell provides you with the capability to match sets of characters using the `[]` wildcard.

Matching sets of characters II

- The syntax for using this wildcard is:
command [characters]
- Here *command* is the name of a command, such as **ls**, and *characters* represents the characters you want to match.
- The following commands fulfill the previous requirements:

\$ ls ch0[0123456789].*

o

\$ ls ch0[0-9].*

- The following example lists all the files starting with a lowercase letter:

\$ ls [a-z]*

Matching sets of characters III

- The `[]` wildcard also enables you to combine sets by putting the sets together. For example:

```
$ ls [a-zA-Z]*
```

matches all files that start with a letter, whereas the command:

```
$ ls *[a-zA-Z0-9]
```

matches all files ending with a letter or a number.

- As you can see from the previous, the maximum amount of flexibility in filename substitution occurs when you couple the `[]` wildcard with the other wildcards.

Negating a set I

- Consider a situation where you need a list of all files except those that start with the letter *a*.
- You have two approaches to solving this problem:
 - ① Specify all the characters you want a filename to contain.
 - ② Specify that the filename should not include the letter *a*.
- If you choose the first approach, you need to construct a set of all the characters that your filename can contain. You can start with:

`[b-zA-Z0-9]*`

- This set does not include the special characters that are allowed in filenames.

Negating a set II

- Attempting to include all these characters creates a huge set that requires complicated quoting. An approximation of this set is:

```
[b-zA-Z0-9\-\_ \+ \= \\ \\' \" \{ \} \] *
```

- Compared to this, the second approach is much better because you only need to specify the list of characters that you do not want.
- The `[` wildcard provides you the capability to match all characters except those that are specified as the set.
- This is called **negating the set**, which you can accomplish by specifying the `!` operator as the first character in a set. The syntax is:

```
command [!characters]
```

Negating a set III

- Here, *command* is the name of a command, such as **ls**, and characters is the set of characters that you do not want to be matched.
- For example, to list all files except those that start with the letter *a*, you can use the command:

```
$ ls [!a]*
```

Variable substitution I

- Variable substitution enables the shell programmer to manipulate the value of a variable based on its state.
- Variable substitution falls into two categories:
 - ① Actions taken when a variable has a value.
 - ② Actions taken when a variable does not have a value.

Here is a summary of all variable substitution methods:

- **`${parameter:-word}`**: If *parameter* is null or unset, *word* is substituted for *parameter*. The value of *parameter* does not change.
- **`${parameter:=word}`**: If *parameter* is null or unset, *parameter* is set to the value of *word*.

Variable substitution II

- **`${parameter:?message}`**: If *parameter* is null or unset, *message* is printed to standard error. This checks that variables are set correctly.
- **`${parameter:+word}`**: If *parameter* is set, *word* is substituted for *parameter*. The value of *parameter* does not change.

Substituting a default value

- The first form enables a default value to be substituted when a variable is unset or null.
- This is formally described as:

`${parameter:-word}`

Here *parameter* is the name of the variable, and *word* is the default value.

- A simple example of its use is:
`PS1=${HOST:-localhost}"$ " ; export PS1 ;`
- You could use this in a user's .profile to make sure that the prompt is always set correctly.
- This form of variable substitution does not affect the value of the variable. It performs substitution only when the variable is unset.

Assigning a default value |

- To set the value of a variable, the second form of variable substitution must be used.
- This form is formally described as:

`${parameter:=word}`

Here, *parameter* is the name of the variable, and *word* is the default value to set the variable to if it is unset.

- Appending the previous example, you have:
`PS1=\${HOST:=`uname -n`} "\$ " ; export PS1 HOST ;`
- After the execution of this statement, both `HOST` and `PS1` are set.

Assigning a default value II

- This example also demonstrates the fact that the default string to use does not have to be a fixed string but can be the output of a command.

If this substitution did not exist in the shell, the same line would have to be written as:

```
if [ -z "$HOST" ] ; then
    HOST=`uname -n` ;
fi ;
PS1="$HOST$ "; export PS1 HOST ;
```

- As you can see, the variable substitution form is shorter and clearer than the explicit form.

Aborting due to variable errors I

- Sometimes substituting default values can hide problems.
- The *sh* supports a third form of variable substitution that enables a message to be written to standard error when a variable is unset. This form is formally described as:

`${parameter:?message}`

- A common use of this is in shell scripts and shell functions requiring certain variables to be set for proper execution.
- For example, the following command exits if the variable `$HOME` is unset:
`: ${HOME:?}"Your home directory is undefined."}`

Aborting due to variable errors II

- In addition to using the variable substitution form described previously, you are also making use of the no-op (no-op as in no operation) command, `:`, which simply evaluates the arguments passed to it.
- Here you are checking to see whether the variable `HOME` is defined. If it is not defined, an error message prints.
- The final form of variable substitution is used to substitute when a variable is set. Formally this is described as:

`${parameter:+word}`

Here *parameter* is the name of the variable, and *word* is the value to substitute if the variable is set.

- This form does not alter the value of the variable; it alters only what is substituted.

Aborting due to variable errors III

- A frequent use is to indicate when a script is running in debug mode:

```
echo ${DEBUG:+"Debug is active."}
```

Command and arithmetic substitution

Two additional forms of substitution provided by the shell are:

- 1 **Command substitution:** enables you to capture the output of a command and substitute it in another command.
- 2 **Arithmetic substitution:** enables you to perform simple integer mathematics using the shell.

Command substitution I

- Command substitution is the mechanism by which the shell performs a given set of commands and then substitutes their output in the place of the commands.
- Command substitution is performed when a command is given as: ``command``
Here *command*, can be a simple command, a pipeline, or a list.
- Make sure that you are using the backquote, not the single quote character, when performing command substitution.
- Command substitution is generally used to assign the output of a command to a variable:

Command substitution II

```
DATE=`date`  
USERS=`who | wc -l`  
UP=`date ; uptime`  
grep `id -un` /etc/passwd
```

Arithmetic substitution I

- In *ksh* and *bash*, the shell enables integer arithmetic to be performed. This avoids having to run an extra program such as **expr** or **bc** to do math in a shell script.
- This feature is not available in *sh*.
- Arithmetic substitution is performed when the following form of command is given:

`$((expression))`

- Expressions are evaluated according to standard mathematical conventions.
- Following are the operators listed in order of precedence:

`/, *, -, +, ()`

Arithmetic substitution II

- You use the following command as an illustration of the operators and their precedence:

```
foo=$(( (5 + 3 * 2) - 4) / 2 ))
```

- After this command executes the value of foo is set to 3. Because this is integer arithmetic, the value is not 3.5, and because of operator precedence the value is not 6.

Introduction

- The way the shell performs substitution is generally useful, but sometimes it is necessary to turn off shell substitution and let each character stand for itself.
- Turning off the special meaning of a character is called quoting, and it can be done three ways:
 - ① Using the backslash (\).
 - ② Using the single quote (').
 - ③ Using the double quote (").
- Quoting can be a very complex issue, even for experienced UNIX programmers.
- You will learn a series of simple rules to help you understand when quoting is needed and how to do it correctly.

Quoting with backslashes I

- Here is a list of most of the shell special characters (also called metacharacters):

`* ? [] ' " \ $; & () | ^ < > new-line space tab`

Watch what happens if you add one of them to the echo command:

```
$ echo Hello world
```

```
$ echo Hello; world
```

- The semicolon (;) character tells the shell that it has reached the end of one command and what follows is a new command. This character enables multiple commands on one line.

Quoting with backslashes II

- The backslash causes the ; character to be handled as any other normal character.

```
$ echo Hello\; world
```

- To display any shell special character reliably from echo, you must *escape it*, that is, precede it by a backslash.
- Using the backslash quotes the character that follows it, using it as a literal character.
- Each of the special shell characters previously listed causes a different problem symptom if you try to echo it without quoting it.
- This need to quote special characters occurs in many other UNIX commands.

Using single quotes |

- Here is an echo command that must be modified because it contains many special shell characters:

```
echo <-$1250.**>; (update?) [y|n]
```

Putting a backslash in front of each special character is tedious and makes the line difficult to read:

```
echo \<-\$1250.\*\*\>\; \ (update\?\) \ [y\|n\]
```

- There is an easy way to quote a large group of characters. Put a single quote (') at the beginning and at the end of the string:

```
echo '<-$1250.**>; (update?) [y|n]'
```


Using single quotes II

Any characters within single quotes are quoted just as if a backslash is in front of each character.

- If a single quote appears within a string to be output, you should not put the whole string within single quotes:

```
$ echo 'It's Friday'
```

This fails and only outputs the following character, while the cursor waits for more input:

```
$> _
```

- The `>` sign is the *secondary shell prompt* (as stored in the `PS2` shell variable), and it indicates that you have entered a multiple-line command (what you have typed so far is incomplete).

Using single quotes III

- Single quotes must be entered in pairs, and their effect is to quote all characters that occur between them.
- In case you are wondering, you cannot get around this by putting a backslash before an embedded single quote.
- You can correct the previous example by not using single quotes as the method of quoting.
- Use one of the other quoting characters, such as the backslash:

```
$ echo It\'s Friday
```

Using double quotes I

- Single quotes can sometimes take away too much of the shell's special conveniences.
- The following echo statement contains many special characters that must be quoted in order to use them literally:

```
$ echo '$USER owes <-$1250.**>;  
    [ as of (`date +%m/%d`)]'
```

The output using single quotes is easy to predict—what you see is what you get:

```
$USER owes <-$1250.**>; [ as of (`date +%m/%d`) ]
```

- Single quotes prevent variable substitution, so \$USER is not replaced by the specific user name stored in that variable.

Using double quotes II

- Single quotes also prevent command substitution, so the attempt to insert the current month and day using the date command within backquotes fails.
- Double quotes are the answer to this situation. Double quotes take away the special meaning of all characters except the following:
 - `\$` for parameter substitution.
 - Backquotes for command substitution.
 - `\$` to enable literal dollar signs.
 - `\`` to enable literal backquotes.
 - `\"` to enable embedded double quotes.
 - `\\` to enable embedded backslashes.
 - All other `\`characters are literal (not special).

Using double quotes III

- Watch what happens if you use double quotes like this:

```
$ echo "$USER owes <-$1250.**>;  
[ as of (`date +%m/%d`)]"
```

```
$ echo "$USER owes <-\$1250.**>;  
[ as of (`date +%m/%d`)]"
```

```
$ echo "The DOS directory is \"\\windows\\temp\""
```

Quoting rules and situations I

- Quoting ignores word boundaries:

```
$ echo "Hello; world"
```

```
$ echo Hel"lo; w"orld
```

- Combining quoting in commands:

```
$ echo The '$USER' variable contains this value \>  
"|$USER|"
```

- Embedding spaces in a single argument:

Quoting rules and situations II

```
$ echo Name Address
$ echo "Name Address"
$ mail -s Meeting tomorrow juan pedro
    < meeting.notice
$ mail -s Meeting\ tomorrow juan pedro
    < meeting.notice
$ mail -s 'Meeting tomorrow' juan pedro
    < meeting.notice
$ mail -s "Meeting tomorrow" juan pedro
    < meeting.notice
```

Quoting rules and situations III

- The newline character is found at the end of each line of a UNIX shell script; it is a special character that tells the shell that it has encountered the end of the command line.
- Normally you can't see the newline character.
- You can quote the newline character to enable a long command to extend to the next line:

```
$ cp file1 file2 file3 file4 file5 file6 file7 \  
> file8 file9 /tmp
```

Notice the last character in the first line is a backslash, which is quoting the newline character implied at the end of the line.

Quoting rules and situations IV

- The shell recognizes this and displays `>` (the PS2 prompt) as confirmation that you are entering a continuation line or multiple-line command.
- You must not enable any spaces after the final backslash for this to work. A quoted newline is an argument separator just like a space or tab. Here is another example:

```
$ echo 'Line 1  
> Line 2'
```

- Quoting the special character takes away its wildcard meaning.
- You should always quote your regular expressions to protect them from shell filename expansion