SuSE Linux Enterprise Server
Shell Programming with the Bash Shell

SuSE Linux AG

SuSE Linux AG
Training Document – Article No. 45361-2INT

SuSE Linux Enterprise Server

Authors: Peter Albrecht, Anke Börnig, Berthold Gunreben, Marion Reyzl, Jesko Schneider, Dr. Oliver Schrögel, Dieter Thalmayr, Michael Weyrauch

Release: January 2004 (SuSE Linux Enterprise Server 8)

Translation/Editing: Olaf Niepolt, Steve Tomlin, Tino Tanner

All programs, illustrations and information contained in this manual were compiled to our best knowledge and tested carefully. This, however, does not exclude the possibility of errors. For this reason, the program material contained in this present manual shall not constitute any obligation or guarantee of any kind. The authors of SuSE Linux AG will thus accept no responsibility or in any way be held liable for damages of any kind which may result from the use of this program material, parts thereof, or for any resulting violation of the law by third parties.

The representation of registered names, trade names, the naming of goods etc. in this training manual does not give the right, even where not specifically stipulated, to assume that such names, in terms of trade names or protection of trade name legislation, can be regarded as free and thus be put to use by anybody whosoever.

All trade names are used without the guarantee for their free use and may possibly be registered trade marks. SuSE Linux AG essentially adheres to the guidelines of the manufacturers. Other products named here may be trade marks of a respective manufacturer.

This work is protected by copyright. All rights in connection with the reproduction or copying of this training manual or parts thereof are reserved. This also applies to translations thereof. No part of this work may, in any form whatsoever (print, photocopy, microfilm or any other procedures) and also not for training purposes, be reproduced or electronically processed, duplicated, or disseminated without the written permission of the publisher.

© 2003 SuSE Linux AG

Internet: http://www.suse.de/training/

© 2004, SUSE LINUX AG (http://www.suse.com/training/)

Contents

© 2004, SUSE LINUX AG (http://www.suse.com/training/)

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

© 2004, SUSE LINUX AG (http://www.suse.com/training/)

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

1 Introduction

In this chapter, learn

- what a shell is
- which different types of shells exist
- what the advantages and disadvantages of shell scripts are

© 2004, SUSE LINUX AG (http://www.suse.com/training/)

1.1 Shells

Whenever a user works on a computer, there needs to be an interface between user and operating system or between the operating system and the application to communicate the user's commands to the system. Such an interface may be presented to the user in the form of a graphical desktop, but it may also be a command-line interface. An example for such a command-line interface is the one provided by the `command.com` interpreter under DOS.

On Unix systems, the standard interface between user and system is the shell. Strictly speaking, there is no such thing as **the** shell, because Unix systems tend to support a number of different shell programs.

1.2 Types of Unix Shells

Although Unix-type operating system provide various shells, this chapter will only discuss the characteristics and programming features of the Bash shell. For the sake of completeness, however, we want to mention some of the more common alternatives:

Bourne Shell Named after its developer, Steve Bourne, there are two different variants of this shell :

`/bin/sh` The "original" Bourne shell — the default shell of most operating systems belonging to the Unix family. The Bourne shell is a simple command interpreter and most other shells are an extended implementation of it.

`/bin/bash` **B**o(u)rn(e) **a**gain **sh**ell, which provides extended Bourne shell functionality and is the standard shell on Linux systems. Bash is backward compatible with the Bourne shell, so the latter is often not installed anymore (on the SuSE Linux Enterprise Server, `/bin/sh` is only a link to `/bin/bash`).

C shell The C shell has a syntax similar to that of the C programming language. There are several flavors:

`/bin/csh` The "original" C shell.

`/bin/tcsh` A successor to the C shell with some enhancements.

Korn shell, `/bin/ksh` The Korn shell is named after its developer, David Korn. It provides some C shell features but also some of the features found in Bash (such as the history function).

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

When working on the command line, refer to the variable SHELL to see which shell program you are actually using:

```
tux@earth:~> echo $SHELL
/bin/bash
```

The first line of a shell script includes a statement specifying which shell to use for it. Therefore it is not relevant whether the script itself is started from within another shell or not (see Section 3.3 on page 37).

1.3 Uses for Shell Scripts

Many system administration tasks involve some kinds of chores that

- need to be performed again and again
- are the same for a given set of elements

These tasks are obvious candidates for some kind of automation. Apart from that, entering one command after another by hand is a rather error-prone process. If you mistype one of the commands, you may need enter everything again. This can be solved by storing the entire command sequence in a file, which can then be executed in a much more dependable way.

1.4 Advantages of Shell Scripts

There are many areas where shell scripts do a better job than other kinds of programs. Shell programs have the advantage that they are:

- easy to write
- portable

Shell scripts have a syntax that is easy to grasp and the available commands should already be known from everyday use. Also, a shell script abstracts from the underlying hardware architecture and can therefore run on very different platforms without modifying the code. Finally, scripted code does not require any compiling. This is a very obvious plus at a time when Linux is available for a growing number of hardware platforms.

On the other hand, there are tasks where shell scripts perform rather badly or are useful only to a limited degree. Shell scripts are also rather slow and tend to use lot of CPU power.

All commands of a shell script are processed by a command interpreter — which is the shell program itself — and their execution is therefore comparatively slow. For extensive computations, it is often necessary to rely on external programs, which results in much slower execution compared to Perl scripts or simple C programs. In addition, shell scripts can use a lot of processor power especially when quickly iterating through a larger number of loops.

Shell scripts are a good solution for many tasks, especially those related to system administration. More complex routines, however, may be solved much better with other scripting languages, especially where quick execution is crucial or where high CPU loads are expected because of the nature of the task.

Summary

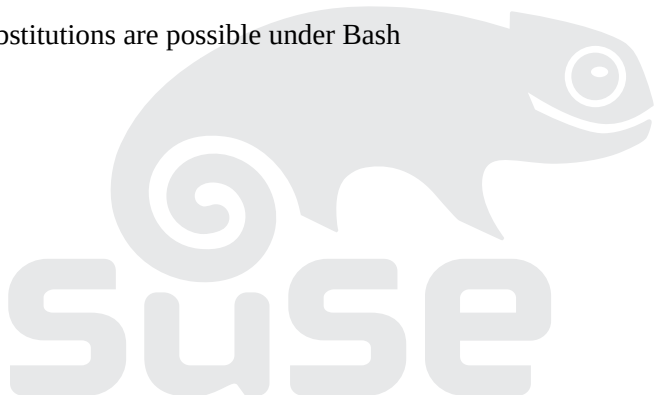
- The shell acts as the interface between the operating system and the user.
- The Bourne shell is the standard shell for Unix in general. Linux uses Bash, an enhanced Bourne shell variant, as its standard command interpreter.
- Shell scripts are a good choice for those tasks that need to be done repeatedly in the same fashion. However, they are relatively slow and can cause a high CPU load.

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

2 Bash Basics

In this chapter, learn

- which are the configuration files of Bash
- how to redirect the standard data channels
- how to use aliases and the command history
- how to use variables
- how to quote special characters
- which substitutions are possible under Bash



2.1 Initialization Files

To customize Bash for an interactive session, it is useful to know about the configuration files and about the order in which they are processed. To understand how things are interrelated, there is one important distinction to be made between two different types of shells: *login shells* and *non-login shells*.

A login shell is started whenever a user logs in to the system. By contrast, any shell started from within a running shell is a non-login shell. The only difference between these two is in the kind of configuration files read by them when started.

A login shell will also be started whenever a user logs in through an X display manager. Therefore, all subsequent terminal emulation programs run non-login shells.

2.1.1 Login Shells

The following files are read when starting a login shell:

`/etc/profile` A system-wide configuration file read by all shells. It sets things like the umask (which determines file permission defaults) and a number of different variables.

`/etc/bash.bashrc` A system-wide configuration file of the SuSE Linux Enterprise Server for Bash-specific settings, such as alias definitions and a special variable to define the form of the command prompt.

`~/.bash_profile` This is the first user-level configuration file that Bash tries to find. It stores user customizations, for example, to change the command prompt. By default, this file is not present on the SuSE Linux Enterprise Server.

`~/.bash_login` The second user-specific file that Bash tries to find. By default, the file is not created on the SuSE Linux Enterprise Server.

`~/.profile` On the SuSE Linux Enterprise Server, this file is created for each new user by default. Any user-specific customizations can be stored in it. It will be read not only by Bash, but also by other Unix shells.

2.1.2 Non-Login Shells

Only one file will be read when a non-login shell is started:

`~/.bashrc` A configuration file read only by Bash in which users can store their customizations.

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

Most Linux distributions have a default setup that ensures users do not see any difference between a login shell and a non-login shell. In most cases, this is achieved by also reading the ~/.bashrc file when a login shell is started.

On the SuSE Linux Enterprise Server, the file /etc/profile includes a command to read in the file /etc/bash.bashrc if it exists, but also the ~/.bashrc file, so the settings in the latter are valid for any login shells, too.

In other words, the system-wide configuration files are read in only once when a user logs in at the system. By contrast, the user-specific configuration file ~/.bashrc is evaluated only when a new instance of the shell is started afterwards.

If you change any settings and want them to be applied during the same shell session, the changed configuration file needs to be read in again. However, this cannot be achieved by simply entering the file name:

```
tux@earth:~> .bashrc
bash: ./.bashrc: Permission denied
```

By entering the file name, you tell the shell that it should try to execute this file and treat it like a command, which will fail because the file does not have execute permissions. Apart from that, even if it were executed, the shell would start a subshell and the changes would only apply to the latter (see Section 2.3 on page 9). The proper way to read in a changed configuration file and to apply the changes to the current session is by using the internal shell command source.

```
tux@earth:~> source .bashrc
tux@earth:~>
```

You may also use the "short form" of this command, which happens to be included in many configuration files, where it is used to read in other configuration files. For instance, in the file /etc/profile, you can find this:

```
test -s /etc/profile.local && . /etc/profile.local
```

What this line does is check whether the file /etc/profile.local exists (for test, see Appendix A.7 on page 91). If the file does exist, it is read in by the "." command. The dot, separated from the file name by a space, is the short form of the source command just mentioned.

To sum this up, in order to apply changes in one of the Bash initialization files immediately, tell Bash to reread the file with source *filename*.

2.2 Aliases

Defining aliases allows you to create shortcuts for commands and their options or to create commands with entirely different names. On a SuSE system, whenever you enter the commands dir , md or ls, for instance, you will be using aliases.

You can find out about the aliases defined on your system with the command `alias`. This will show you that `dir`, for instance, is only an alias for `ls -l` and that `md` is an alias for `mkdir -p`. These two are examples of aliases through which new commands are defined:

```
tux@earth:~> alias | grep mkdir
alias md='mkdir -p'
tux@earth:~> alias | grep dir
...
alias dir='ls -l'
...
```

To see whether a given command is an alias for something else, use the `type` command. For each command specified, `type` will tell you whether it is a built-in shell command, a regular command, or an alias. For regular commands, the output of `type` lists the path to the corresponding executable. For aliases, it lists the elements aliased:

```
tux@earth:~> type -a ls
ls is aliased to `ls $LS_OPTIONS'
ls is /bin/ls
```

The above example shows that `ls`, too, is an alias, although in this case it is only used to add some options to the command.

The `-a` option was used for `type` to show both the contents of the alias and the path to the original `ls` command. The output shows that `ls` is always run with the options as stored in the variable `LS_OPTIONS`. These options cause `ls` to list different file types in different colors (among other things).

Most of the aliases used on a system-wide basis are defined in the file `/etc/bash.bashrc`. Aliases are defined with the `alias` command and can be removed with the `unalias` command. As an example, entering `unalias ls` would remove the alias for `ls`, causing `ls` to stop coloring its output.

The syntax to define aliases is as follows:
`alias` *`aliasname`*`="`*`command options`*`"`

An alias defined in this way is only valid for the current shell and will not be inherited by subshells. To make an alias persistent, you need to store the definition in one of the shell's configuration files.

Example:

```
tux@earth:~> alias ps="echo Hello"
tux@earth:~> ps
Hello
tux@earth:~> bash
tux@earth:~> ps
  PID TTY          TIME CMD
  858 pts/0    00:00:00 bash
  895 pts/1    00:00:00 bash
  ...
```

On the SuSE Linux Enterprise Server, the file `~/.alias` is created for personal aliases defined by each user. This file is read in by `~/.bashrc`, where a command is included

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

to that effect. Aliases are not relevant to shell scripts at all, but can be a real time saver when using the shell interactively.

Exercise

1. Under your user account, create an alias for the `cp` command, such that it is always run with the `-i` option.

2. The command `ssh -X root@localhost` can be used to connect as *root* to the local X server via SSH, provided that X forwarding has been enabled on the host. This allows regular users to start graphical applications as *root*. To abbreviate the above command, create an alias for it under your own user account.

2.3 Variables

The behavior of the shell is largely influenced by its internal (built-in) variables. Users can also define their own variables then use them as needed, for instance, in shell scripts. There is an important distinction to be made between the *shell variables* and the *environment variables*.

Shell variables control the behavior of the shell itself and are only relevant locally (for the currently active shell). Examples for shell variables are `HISTSIZE`, `PS1`, and `UID`.

Environment variables have a larger scope and also influence any other programs started from within the current shell. In other words, they are inherited by any subshells or child processes. Examples for environment variables are `HOME`, `PATH`, `DISPLAY`, and `PWD`.

To understand better that shell variables and environment variables behave differently, it is useful to have a closer look at the way in which the shell executes a command:

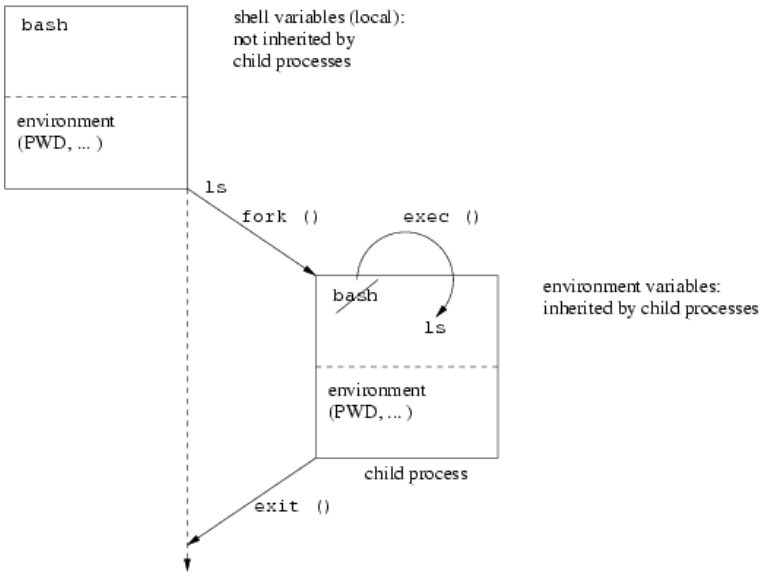


Figure 2.1: ***The Fork & Execute* Mechanism of the Shell**

When an external command is called by the shell, the latter first starts a subshell — it loads an image of itself into memory. The frames in the figure each represent a process as present in memory when this first step has happened. So the process has actually doubled itself, a mechanism called forking. Now when the process is forked, the environment of the parent process (which in our example consists of the `PWD` variable — the name of the current working directory) is inherited by the child process. However, since the actual command that has been called is the `ls` command, the shell replaces the second Bash process with the `ls` process. The shell does so using its internal `exec` call. This is done without changing the original environment variables (such as `PWD`). When the command has finished its job, the process is terminated through the `exit` call and control is handed over to the original shell again, which will always wait for the child process to come to an end.

It should be clear from the figure that changes to the environment of the child process cannot have any influence on the environment of the parent shell.

2.3.1 Working With Variables

The syntax to set a shell variable is `VARIABLE=value`, by which the contents of `value` are directly assigned to the new variable. Being a shell variable, it will not be inherited by any subshells.

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

Example:

```
tux@earth:~> VARIABLE1="Good morning"
tux@earth:~> echo $VARIABLE1
Good morning
tux@earth:~> bash
tux@earth:~> echo $VARIABLE1

tux@earth:~>
```

To refer to an existing value, it must be prefixed with the '$' sign. Thus, the command `echo $VARIABLE` will return the value of that variable.

With the command `export`, a shell variable can be turned into an environment variable. Also use the `export` command to define an environment variable right away. As has been explained above, such a variable will then be inherited by any subshells.

Example:

```
tux@earth:~> export VARIABLE2="Good afternoon"
tux@earth:~> bash
tux@earth:~> echo $VARIABLE2
Good afternoon
tux@earth:~>
```

To see which variables have been set for your shell, you can use these commands:

`export` Lists all environment variables.

`set` Lists all variables as well as functions that have been declared [1].

`env` displays a list of all currently defined variables and their content. [2].

The command `unset` *`variable`* can be used to delete a variable.

Example:

```
tux@earth:~> a=10
tux@earth:~> echo $a
10
tux@earth:~> unset a
tux@earth:~> echo $a

tux@earth:~>
```

Important: When defining variables within shell scripts, always make sure that they are different from any existing variables, such as `UID` or `HOME`. Otherwise, you may have to deal with conflicts and error messages.

[1] Moreover, `set` *`variable value`* can be used to modify the shell attributes during runtime. Check the bash man page (`man 1 bash`) for more information

[2] Moreover `env` *`variable=value command`* can be used to execute commands in a modified environment (i.e., with specially set variables). Check the man page (`man 1 env`) for more information

In summary, if you want a variable to take effect on a local basis only, you should set it as a shell variable. If you want a variable to be inherited by subshells as well, use `export` to define it as an environment variable.

2.3.2 Important Internal Bash Variables

Knowing about the important internal Bash variables is very useful when customizing the shell for interactive sessions as well as for shell scripting. They are listed in the following table:

Variable	Description
HOME	the user's home directory
PATH	the search path for commands
PWD	current working directory
IFS	the internal field separator; i.e., the character that separates individual arguments from each other
PS1	the primary shell prompt
PS2	the secondary shell prompt
PS3	the tertiary shell prompt (see `select`)
?	the exit status or (return value) of the most recent child process
$	the process ID of the current shell itself
#	the number of arguments passed to the shell
0-9	argument 0 (usually the command itself), argument 1, and so on, as passed to the shell
*	all arguments (with the exception of argument 0) as a single word or argument
@	all arguments (with the exception of argument 0) as separate words or arguments

Exercises

Exercise 1

1. Create a shell variable called `USERNAME` that holds your user name. Then use the `su` command to log in as *root*. What are now the contents of the variable?
2. Log out from the *root* shell to return to your normal user shell. Convert the variable into an environment variable using the `export` command. Now use `su` again to log in as *root*. What are the contents of the variable this time?

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

3. Change the primary Bash prompt (variable `PS1`) such that it displays the current time or the process ID of the last process, for instance. Do this so the changes take effect immediately.

Exercise 2

1. As a regular user, execute the `ls` command in your home directory then check the exit status.
2. As a regular user, execute the `ls` command for the home directory of user *root* (`/root`) then check the exit status.

Exercise 3

1. Enter the command `bash -s 1 2 3`. What do you think will be the output of `echo $#`?
2. What is the command to obtain the third argument of the above command?
3. How can you obtain the above command without any of its arguments?

2.4 Command History

When used interactively, Bash keeps a list of all the commands that have been entered in the command history. The variable `HISTSIZE` determines how many commands are kept in the history and the variable `HISTFILE` determines to which file it is saved. The default history file is `~/.bash_history` and the default history size is set to 500.

The history file is read in each time a shell session is opened. All commands entered by the user are added to the existing command history. However, these additions are only written to the file `~/.bash_history` when closing the shell session. Therefore, the history tends to be different from shell to shell when running several shells in parallel.

There are several ways to access the history. The command `history` prints a numbered list of all the commands stored in the history. You can use the numbers to access a particular command with `!`*n*. This causes the command with number *n* to be executed again — you are not prompted to edit the command before execution.

```
tux@earth:~> history
    1  ls
    2  ps ax
    3  echo $SHELL
    4  history
tux@earth:~> !3
echo $SHELL
/bin/bash
tux@earth:~>
```

To reexecute the last command in the history, enter `!!`. In addition, browse through the history list simply by pressing ↑ to go back in the list and ↓ to go forward. The latter method will just display the corresponding command, so that you can edit it before re-execution.

It is also possible to search for commands in the history. By entering `!`*`string`*, directly execute the last command started with *`string`*. Here, again, it is not possible to edit the command before execution. On the other hand, you can search for commands when browsing through the history list: just enter one or two letters then Page↑ or Page↓ to jump to the previous or next command in the history beginning with these letters.

There are other possibilities to access the command history, but the most important ones are the use of '!' to directly rerun a previously entered command and the arrow keys to manually browse the history for the right command.

2.5 Input and Output Channels

Each programs relies on two standard data channels to receive and to print data plus another channel to output error messages.

The conventional names of the three standard data channels are:

- standard output — stdout
- standard input — stdin
- standard error (output) — stderr

When a program is started from the shell, all three of these data channels are connected to the current terminal: all output (standard output and standard error) goes to the screen and all standard input is read from the keyboard.

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

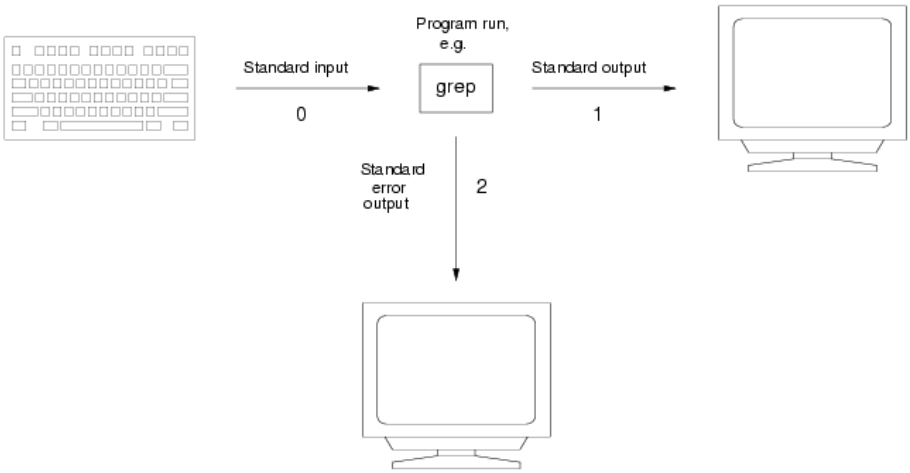


Figure 2.2: **Standard Data Channels**

Each of the three data channels has a number:

standard input (stdin)	0
standard output (stdout)	1
standard error (stderr)	2

These numbers can be referenced to redirect the data channels and each of the three I/O channels can be redirected independently from the others. Redirection allows you to write to a file (output, errors), but also to read from a file (input). Redirection also allows you to link up two processes by feeding the output of the first process through a pipe then so the second process takes this as its input.

2.5.1 Redirection to Files

All three data channels can be redirected to files.

A program could read the required parameters from a file instead of the standard input. The word count command `wc`, for instance, counts the number of lines, words, and characters in any input supplied to it. By default, the command expects some input on stdin:

```
tux@earth:~> wc
```

The `wc` program will stop reading from the input only when Ctrl + D are pressed.

If you type something before stopping, `wc` will print the number of lines, words, and characters:

```
tux@earth:~> wc
Once upon a midnight dreary,
while I pondered, weak and weary, (...)
E.A. Poe
      3       14      78
tux@earth:~>
```

The input can also be taken directly from a file. This requires the '<' operator to redirect the input:

```
tux@earth:~> wc < /etc/passwd
     42      93    2199
tux@earth:~>
```

What the '<' operator does is to tell the program to read its input from the specified file, rather than from standard input. However, most commands are able to directly use a file as their input even without any redirection. In other words, most commands that normally take a file name as an input argument are also able to directly read from stdin.

```
tux@earth:~> wc /etc/passwd
     42      93    2199
```

The above example shows that a file can be specified as the input for `wc` without having to use the '<' for stdin redirection. In a similar manner, you can redirect standard output (which normally goes to the screen) so it is written to a file.

```
tux@earth:~> date
Mon Sep 16 11:16:03 CEST 2002
tux@earth:~> date > timestamp
tux@earth:~>
```

In this example, the output of the `date` command is not sent to standard output but to the file `timestamp`.

Important: If the file to which output is redirected does not exist yet, it is automatically created. More seriously, if a file of the same name already exists, it is overwritten without prompting you for confirmation.

To avoid that an existing file is overwritten or to write several output strings to the same file, use the append operator '>>'.

```
tux@earth:~> date > timestamp
tux@earth:~> date >> timestamp
tux@earth:~>
```

After executing these commands, the `timestamp` file should contain two separate lines with the output of `date`.

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

All error messages are put out on a separate channel, which by default points to the screen as well. However, the next example shows that stdout and stderr are indeed handled separately:

```
tux@earth:~> ls / /root
/:
bin   dev  home  media  opt   root  srv  usr
boot  etc  lib   mnt    proc  sbin  tmp  var

ls: /root: Permission denied
tux@earth:~> ls / /root > outputfile
ls: /root: Permission denied
```

As you can see, the error message is still printed to the screen, while the `/` list is redirected to `outputfile`.

The format to redirect the standard error output to a file is '`2> filename`'. Again, if the file already exists, it will be overwritten without asking you for confirmation. Just like with stdout, you can append the error output to a file with '`2>>`'.

```
tux@earth:~> ls / /root 2> errorfile
/:
bin   dev  home  media  opt   root  srv  usr
boot  etc  lib   mnt    proc  sbin  tmp  var
tux@earth:~> ls / /root > outputfile 2> errorfile
tux@earth:~>
```

The first command only redirects the error output to the `errorfile`. The second command redirects both stdout and stderr to two different files.

It is also possible to redirect both channels into the same file. To achieve this, one channel needs to be redirected to the other one. One possibility is to redirect standard error to standard output with "`2>&1`". The other one is to redirect standard output to standard error with "`1>&2`".

```
tux@earth:~> find / -name "*.txt" 2>&1
```

In this command, the error output is redirected to stdout.

To write all the output to a file, we still need to specify a file name. Here it is important to observe the correct order: First, stdout needs to be redirected to a file. This is followed by the stderr to stdout redirection, like in this command:

```
tux@earth:~> find / -name "*.txt" > files-found 2>&1
```

So the output of the `find` command is written to a file called `files-found` and subsequently any error output is redirected to standard output.

As a simpler method to redirect all output of a command, use the "`&>`" operator. One example where this comes in very handy are `cron` jobs, where it is useful to discard whatever output is produced by them:

```
42 * * * * somecommand &> /dev/null
```

With this crontab entry, `somecommand` would be executed 42 minutes past every full hour. All output is effectively sent to "nowhere" — to the data sink `/dev/null`.

In some cases, you may also want to redirect any standard output to the stderr channel. For instance, if a shell script was written such that error messages are printed with the `echo` command, they are actually put out on stdout. To change this, one could redirect them to standard error:

```
echo "This is an error message in a script," 1>&2
echo "which bothers us on standard output, so" 1>&2
echo "we send it to standard error instead." 1>&2
```

If redirected as above, the error messages can be intercepted and dealt with directly on stderr, which could then be redirected again.

In summary, there are the following possibilities to redirect the shell's standard input, standard output, and standard error to a file:

`< filename`	standard input is read from a file
`> filename`	standard output is redirected to a file
`>> filename`	standard output is appended to a file
`2> filename`	standard error is redirected to a file
`2>> filename`	standard error is appended to a file
`2>&1`	standard error is redirected to standard output
`1>&2`	standard output is redirected to standard error
`&> filename`	both standard output and standard error are redirected to a file

2.5.2 The `here` Operator <<

The `here` operator "`<<`" makes it possible to feed more than one line of input to commands that read their input from the command line. Normally, for most of these commands the input stops with the ⏎ key. With the `here` operator, the end of input is reached only when a previously defined delimiter word is entered, so all input up to the delimiter can be read from the command line. This operator is also often used in shell scripts to print several lines to the screen without having to use an `echo` command for each new line.

Examples:

```
tux@earth:~> cat << THE-END
> first input line
> ...
> last input line
> THE-END
first input line
...
last input line
```

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

In this example, the input for the `cat` command is read from standard input until the end delimiter (`THE-END` in our case) is entered at the **beginning** of a new line. This is much less cumbersome than using multiple `echo` commands for the output of several lines.

```
tux@earth:~> sort << EOF
> cherry
> plum
> apple
> banana
> EOF
apple
banana
cherry
plum
```

The `sort` command normally takes a file name as one of its arguments then arranges the input coming from that file in an alphabetical list. In the above example, we use the `here` operator to feed the command with a list of words directly from the keyboard.

Just like in the example, the `here` operator can be used in all those situations where it is necessary to feed more than one line of input to a command that reads input from the command line.

The operator '`<<-`' is a variant of the `here` operator '`<<`'. It removes tabs at the beginning of each line. Tabs can be used to arrange shell scripts in a clear structure. The following program line removes the formatting tabs (see Sample Script 19 in Appendix E on page 118).

```
cat <<- EOF
```

Attention: '`<<-`' only removes tabs. It does not remove spaces.

2.5.3 Feeding Output to Another Process

There are many situations where one might want to link up two programs such that the output produced by a first program is read as the second program's input. For instance, you may want to use the `less` pager to directly process the output of `ls -l`.

One way to achieve this would be to direct the output of `ls -l` to a file, which can then be displayed with `less`:

```
tux@earth:~> ls -l > filelist
tux@earth:~> less filelist
```

However, this can also be accomplished in a more direct way through a pipe. A pipe, represented by the '|' symbol, can read the output of a process as available on stdout and directly feed it to another process on standard input.

Specimen copy

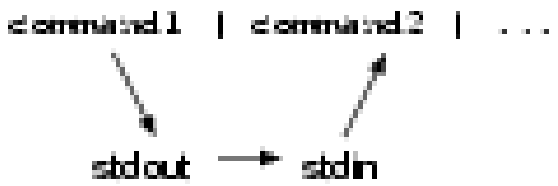


Figure 2.3: **Feeding Standard Output through a Pipe**

To link up the two commands of the above example, simply enter:

```
tux@earth:~> ls -l | less
```

To allow this kind of redirection, the second process must be able to read from standard input.

More examples:

```
tux@earth:~> man -t bash | lpr
```

In this command, the manual page of Bash is converted to PostScript then directly piped to `lpr` for printing.

```
tux@earth:~> ps aux | less
```

As the output of `ps aux` tends to be too long to display on one screen, it is piped to `less` to view it page by page.

This kind of command chaining is often used in shell scripts, as there are many cases where a given string or value needs to be processed several times to be useful for further steps.

Chaining Commands With `xargs`

The Bash shell has a limited input buffer, so there are cases where the list of arguments is simply too long to be passed to a given command. If that happens, the command aborts with an error. This is where the `xargs` command comes to the rescue. The syntax of `xargs` assumes that it receives a list of arguments from a first command through a pipe. The `xargs` command also needs to be told about the program to which it should pass the arguments of the first. Thus, the second command is an argument of `xargs` itself:
command1 | xargs *command2*

The buffer of `xargs` is larger than the shell's buffer. So `xargs` first collects all arguments passed until its buffer is full, then it calls the second command to supply these arguments to it. Then `xargs` collects more arguments from the first command and so on. With its

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

larger buffer, `xargs` is able to feed commands just the morsels that they can swallow. The command is also very useful in other situations.

As has been mentioned, chaining commands through a pipe requires that the second is able to read from standard input. Not all programs have that capability.

Example:

```
tux@earth:~> find . -name "*.old" | rm
rm: too few arguments
Try ‘rm --help’ for more information.
```

With the above command, we intended to use the `find` command to search for all files that are backup copies then to delete them with `rm`. However, this failed because `rm` does not read from standard input.

The `xargs` command can solve the problem by making sure the list of files found is passed as arguments to `rm`.

```
tux@earth:~> find . -name "*.old" | xargs rm
```

The same could be achieved by using the `exec` option of `find`:

```
tux@earth:~> find . -name "*.old" -exec rm {} \;
```

Still `xargs` has the edge over this solution because of another detail: If the files are deleted using the `exec` function of `find`, the `rm` command is called once for each and every file that has been found. By contrast, `rm` will be executed only a few times if called by `xargs`. This will speed up things quite a bit.

Further examples:

```
tux@earth:~> ls *.c | xargs -n1 -i cp {} backups/{}.bak
```

All the files in the current directory with the extension `.c` are passed to the `cp` as separate arguments (option `-n1`) with `cp` creating a copy of each file in the directory `backups`, giving them the extension `.bak`.

```
tux@earth:~> find /etc -type f | xargs grep text
```

This command searches for all regular files in the `/etc` directory and, with the help of `xargs`, the search result is filtered to display only those that contain the string `text`. This is much faster than running the `grep` command on each file found as in `find /etc -type f -exec grep text {} \;` because with `xargs` each run of the `grep` command processes an entire list of files.

2.5.4 Duplicating the Output with `tee`

As soon as the ">" operator is used to direct the output of a command to a file, it becomes impossible to watch the live progress of the command on screen. So a program is needed

that writes to standard output and to a file at the same time. This is what the `tee` command does. In the following example, the output of the first command is piped to `tee`. A file name is specified as an argument for `tee` to tell it to which file the second output stream should be written:

```
tux@earth:~> find / -type f -name "*.c" | tee cfiles-found
```

So in this case it is possible to view the output of the `find` command as it searches through the file system while also having the same output saved to the file `cfiles-found`.

Exercises

Exercise 1

1. Redirect the output of the command `df -h` to a file.
2. Use the `mail` command to send a mail to *root*, which contains the output file of the above exercise.

Exercise 2

1. Mail the output of the command `df -h` to *root* directly — without the intermediate step of saving the command output to a file.
2. Find all files on your system with the executable bit set and display the result with the help of `less` while suppressing all error messages.

2.6 Types of Commands

Entering a command in a shell causes it to be executed, but there are different types of commands and for each of them the execution mechanism is different. The commands `exec` and `export` discussed in Chapter 2.3 on page 9 are examples of the type of commands called shell built-ins. These built-in commands are an integral part of Bash itself and as such they run as part of the current shell process when executed. This means that no child process is started for them.

By contrast, regular programs are external to the shell and exist as real executables on the file system. The fork and execute mechanism as illustrated by Figure 2.3 on page 10 is performed whenever such a program is run.

When a command is called from the shell, the latter first checks whether it is an alias (see 2.2 on page 7) to something else. If that is not the case, the shell checks whether it is a shell built-in. If that is not the case either, the shell will look for it in the program path (as set by the `PATH` variable). This whole search order becomes important if there is a

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

built-in with the same name as a regular command, for instance. If that is the case and if the shell is supposed to execute the regular command, the latter needs to be supplied to the shell together with its complete path.

To check the type of a given command, use the `type` command.

```
tux@earth:~> type cd
cd is a shell builtin
tux@earth:~> type wc
wc is /usr/bin/wc
tux@earth:~> wc

tux@earth:~> type wc
wc is hashed (/usr/bin/wc)
```

For each program specified, `type` informs you whether it is an alias, a built-in, or a command external to the shell. In the case of external commands, `type` also prints the path to the program. Bash remembers whether an external command has already been called from the current shell and `type` returns the message `is hashed` if that is the case. To see all the available information (e.g., the path to the real program hiding behind an alias), enter `type -a`:

```
tux@earth:~> type -a ls
ls is aliased to ‘ls $LS_OPTIONS’
ls is /bin/ls
```

The information about the command type can also be important when looking for help on a given command. For instance, if you want to read help on the `export` command by opening the corresponding manual page (man page) , `man export` will open the manual page of Bash, which contains more than a thousand lines of text. For built-ins, it is more convenient to use the `help` command instead, which provides a short description of the available options.

```
tux@earth:~> help export
export: export [-nf] [name ...] or export -p
NAMEs are marked for automatic export to the environment of
subsequently executed commands.  If the -f option is given,
the NAMEs refer to functions.  If no NAMEs are given, or if ‘-p’
is given, a list of all names that are exported in this shell is
printed.  An argument of ‘-n’ says to remove the export property
from subsequent NAMEs.  An argument of ‘--’ disables further option
processing.
```

2.7 Quoting

Every command line entered in a Bash shell will be interpreted by the Bash program first. This includes any characters with a special meaning to Bash.

These special characters include i. e. spaces, the internal field separator — the character separating individual arguments, and the "$" sign, which is used for variable substitution.

However, there are many cases where one would not want these characters to be evaluated by the shell. For instance, in the case of files names with spaces in them, it is seldom desirable that the space-separated parts are interpreted as separate arguments. To protect special characters from being interpreted by the shell, they need to be quoted or masked. There are different forms of quotation with each of them suitable for different situations:

- \ quotes the following character
- "..." quotes any special characters enclosed with the exception of "$"
- ´...´ quotes any special characters enclosed

Accordingly, if you want to protect just one special character from being interpreted as such by the shell, put a "\" in front of it:

```
tux@earth:~> cp new\ file directory/
tux@earth:~> find / -iname "*.mp3" -exec mv {} /MP3 \;
```

The `find` command, when run with the `exec` option, needs to have a "'`;`"' as its last argument. The same character has a special meaning for the shell, so it needs to be quoted. Similarly, the search string needs to be quoted to make sure that it always is evaluated by `find` and not by the shell.

If you want variables to be interpreted even within a quoted string, enclose the string in double quotation marks ("..."). If do not want the shell to interpret any special characters at all, enclose the string in single quotation marks (´...´).

```
tux@earth:~> echo $HOME
/home/tux
tux@earth:~> echo "$HOME"
/home/tux
tux@earth:~> echo '$HOME'
$HOME
```

Exercise

1. Use the command `touch new\ file.c` to create the file `new file.c` in your home directory.
2. Execute the command `find ~ -name *.c` to search for all C source files in your home directory. If no files are found, what could be the reason?
3. Set the shell to debug mode with `set -x` (see Appendix D on page 99). Then enter the previous command again.
4. How does the `find` command have to look in order to make sure that the C source files in your home directory are found?

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

2.8 Substitution and Expansion

When executing a command in the shell, the entire command line is evaluated by the shell while also interpreting any special characters. The latter includes various forms of substitution and expansion — among other things the shell replaces variables with their values and expands metacharacters to file names (which is also called file globbing). This section discusses the different forms of substitution and expansion.

2.8.1 Variable Substitution

Whenever the shell encounters a "$", it assumes a variable will follow then replaces the variable with its value. Although there is a convention to define variable names in upper-case letters only, this is not strictly required.

This is an example for variable substitution:

```
tux@earth:~> echo $USER
tux
```

2.8.2 Filename Expansion (File Globbing)

When specifying a file name, it is possible to use wild cards or metacharacters representing characters to match — they are evaluated by the shell to expand them to real file names as possible.

The shell can use different wild cards to perform this kind of expansion or file globbing (details can be obtained with `man 7 glob`). However, some of the shell's wild cards for filename expansion are different from those used in regular expressions with a number of other programs (see Appendix B on page 95).

These are the metacharacters used by the shell for filename expansion:

Metacharacter	Expands to / Matches
?	any single character (except "." as the first character of a name as well as "/")
*	any string (including null characters, but except "." as the first character of the name, as well as '/')
[...]	any of the characters enclosed
[!...]	none of the characters enclosed

Examples:

```
tux@earth:~> ls file?
file1  file2  file3
tux@earth:~> ls file*
file  file1  file10  file2  file3
tux@earth:~> ls file[0-9]
file1  file2  file3
tux@earth:~> ls file[0-9]*
file1  file10  file2  file3
tux@earth:~> ls file[!1-2]
file3
```

The first command only lists files that have exactly one character after `file` in their names, since the question mark matches any single character. Also, neither "?" or "`*`" cover a dot at the beginning of a file name, which means that hidden files would not be listed by the first two commands. The "`*`" wild card matches any string including null characters (i.e., no character), so `file` is included in the output of `ls file*`.

A pair of square brackets can be used to specify ranges of characters to match. However, from the character range as enclosed in brackets, only single characters are matched. Accordingly, `file[0-9]` does not match `file10` because this name has two characters after `file`. By adding the "`*`" to the brackets, the shell also expands to all file names with an arbitrary number of the characters from the range. The "!" is a negation metacharacter; it specifies that any single character that does not belong to the specified range should be matched.

There are even more possibilities to use metacharacters for filename expansion.

Brace Expansion

Curly braces allow you to specify a set of characters from which the shell will automatically form all possible combinations. This feature is useful when creating whole sets of files or directories, for instance.

To make this work, the characters to be combined with the given string must be specified as a comma-separated list with no spaces. However, while square brackets allow the use of a hyphen to specify an entire range, this is not possible within curly braces.

Examples:

```
tux@earth:~> touch file{1,2,3}
tux@earth:~> ls
file1  file2  file3
tux@earth:~> mkdir directory{1,2,3}{a,b,c}
tux@earth:~> ls
directory1a  directory1c  directory2b  directory3a  directory3c  file2
directory1b  directory2a  directory2c  directory3b  file1              file3
tux@earth:~> touch file{a-z}
```

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

```
tux@earth:~> ls
directory1a  directory2a  directory3a  file1  file{a-z}
directory1b  directory2b  directory3b  file2
directory1c  directory2c  directory3c  file3
```

Tilde Expansion

In scripts and commands, it is often necessary to specify the home directory of a user. The home directory normally defaults to `/home/`*`login-name`*. This is the directory where each user can store any personal data. However, home directories may be located in other places as well, depending on the system setup. For instance, the home directory of *root* is not a directory under `/home`, but the directory `/root` itself.

With the help of the tilde ("~"), the shell can refer to a user's home directory. The tilde can be used in two different ways:

- ~ Used as a single character, the tilde expands to the home directory of the user running the shell.
- ~***login-name*** If followed by a user (login) name, the tilde is expanded to the home directory of that user without the shell having to know the actual path.

Thus, when using the shell interactively, any user can refer to his home directory by specifying the path as a simple "~".

Examples:

```
tux@earth:/tmp> cp somefile ~/somedirectory

earth:~ # cp /tmp/somefile ~tux
```

The ~*`login-name`* format is especially useful for shell scripts, because it allows referring to the home directory of the actual user that is running the script.

Example:

```
cp $FILE ~$USER
```

Hint: It is often useful to test how metacharacters would be interpreted by the shell before actually executing a command, especially in the case of filename expansion. This can easily be done with the `echo` command. When putting the `echo` command in front of the actual command, the shell will perform all expansions, such that it prints the actual command that would be performed.

```
earth:~ # echo cp /tmp/somefile ~tux
cp /tmp/somefile /home/tux
earth:~ # echo rm -r .[!.]*
rm -r .Xauthority .bash_history .exrc .gnupg .kde .qt .viminfo .xinitrc
```

2.8.3 Command Substitution

There is often a need to process the output of some command as an argument of another command. This can be done through command substitution. Normally, whenever a command is entered, the shell processes the command itself and assumes the following input fields are arguments of that command:

```
tux@earth:~> echo "It's date +%b-%d today."
It's date +%b-%d today.
tux@earth:~> FELLAS=who
tux@earth:~> echo $FELLAS
who
```

In the first command, `date` is only processed as an argument of `echo`. The result is that instead of returning a phrase telling us the current date, the command only echoes the name of the `date` command together with its arguments. In a similar manner, instead of the output of the `who` command, the `FELLAS` variable is assigned only the name of the `who` command.

There are two possibilities to make sure the shell really processes the output of another command in a given command line.

- **$(*command*)** Bash-style command substitution
- **`*command*`** command substitution, Bourne shell–compatible [3]

The two formats are basically interchangeable. However, if you do need Bourne shell compatibility, you should use the second one.

Example:

```
tux@earth:~> echo "It's ‘date +%b-%d‘ today."
It's Dec-11 today.
tux@earth:~> FELLAS=$(who)
tux@earth:~> echo $FELLAS
tux :0 Dec 11 08:36 (console) tux pts/0 Dec 20 08:36
```

Command substitution makes it possible to replace commands with their output, something especially useful in complex command sequences.

2.8.4 Arithmetic Substitution

There are many cases where one would want to perform simple calculations with the values returned by variables. The Bash shell lets you do this through its own, built-in arithmetic engine, although it is limited to integer (whole number) calculations. There is no such

[3] The accent symbol can be set with ⇧ + `.

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

feature in the Bourne shell, so you may need to rely on external commands (such as `expr`).

There are two different formats for arithmetic substitution:

- **`$[ INTEGER1 + INTEGER2 ]`** An arithmetic operation with one of the operators "+", "-", "*", or "/" or a logical operation, such as AND, OR, and others. (This format is being phased out and is therefore not recommended.)
- **`$(( INTEGER1 - INTEGER2 ))`** An arithmetic operation with one of the operators "+", "-", "*", or "/"; other operators are available for logical AND, comparison, and others.

Within such an arithmetic expression, variables can be given without a leading "$".

If the external `expr` command is used, only the four fundamental operations (+, -, *, /) can be performed. The Bash shell, however, allows a much wider scope of operations, including logical AND and comparisons, with the available operators being identical to those of the C programming language. Operators are also described in the Bash manual page (`man bash`) under the heading "ARITHMETIC EVALUATION".

Example:

```
COUNTER=0
...
COUNTER=$(( COUNTER + 1 ))
```

When declaring variables, they can be given a certain attribute. Variables are declared using the `declare` command[4].

If all variables involved in a calculation have been declared as integers, they can also be referenced directly:

```
declare -i INTEGER1
declare -i INTEGER2
declare -i SUM
...
SUM=INTEGER1+INTEGER2
```

2.9 Command-Line Interpretation

Before a command line is executed by the shell, a number of actions are taken to interpret it. This includes the evaluation of special characters, the necessary substitutions or expansions, and others. These steps are performed in a fixed order. Knowing about this order is often useful, which is why it is discussed in this section.

[4] Bash also provides the `typeset` command, which has been adopted from the Korn shell and provides the same set of features as `declare`. However, `typeset` is now considered deprecated, so `declare` should be used wherever possible.

When processing a command line, the shell interprets it in this order:

1. The command line is parsed up to the first command separator, for example, "`;`" or "&&" (see 2.9.1 on the next page).
2. The shell splits the command line into individual words (also called tokens). Words are separated by the internal field separator, or IFS, which by default is a blank (space, tab, or new line).
3. Variables are expanded to their values.
4. The shell executes commands marked for substitution with `...` or $(...), replacing them with their output.
5. Input and output redirection is performed.
6. Any new variables are assigned.
7. Special characters (*, ?, [...]) are interpreted and matched file names are substituted.
8. The command is executed.

This is an example to illustrate how the procedure works:

```
tux@earth:~> echo files: ‘ls $HOME‘ \; date: $(date) > outputfile
```

First, the command line is parsed up to the first separator, in our case the ">" redirecting the output to `outputfile`. The shell then breaks up the command line into individual words, which are "`echo`", "`files:`", "`‘ls $HOME‘`", "`\;`", "`date:`", and "`$(date)`".

The next step is to expand any variables. In the example, this means the variable `HOME` is replaced with its value — `/home/tux`:

```
echo files:  ‘ls /home/tux/‘ \; date:  $(date)
```

This is followed by command substitution — both `ls /home/tux/` and `date` are replaced with their output:

```
echo files:  file1 file2 file3  date:  Mon Nov 11 10:40:43
CET 2002
```

The output of this command line is not printed on standard output, but directed to the `outputfile`. The semicolon is quoted so it appears in the final output instead of being interpreted by the shell.

There are no variables in the above command that would have to be assigned.

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

Finally, the command is executed, with the following output being written to the `outputfile`:

```
files:  file1 file2 file3 ; date:  Mon Nov 11 10:40:43 CET
2002
```

When typing a command line, you should always pay attention to the point at which special characters are going to be interpreted. Do you want the shell to substitute a "`*`" so it expands to a file name? Or is it actually the command itself that is supposed to interpret the asterisk?

There are several ways to debug shell commands. One is to put an `echo` in front of the line, which will print the command line with all the required substitutions filled in. Another one is to set the `-x` option of the shell (with `set -x`), which adds information on the command in its actual form upon execution — again with all the substitutions that are being performed (see Appendix D on page 99).

2.9.1 Command Separators

It is often useful to combine several commands in just one command line. To do so, commands need to be separated from each other, which can be done in several ways.

One possibility is the pipe (see 2.5.3 on page 19) by which the standard output of a first command is directed to the standard input of a second command.

```
tux@earth:~> ls -l /usr/share/doc/packages/ | less
```

Commands can also be combined such that they are executed in a sequence, without their input and output streams being connected to each other. For instance, the execution of a second command can depend on the success or failure of the first one or the second command can be executed without depending on such success or failure.

- **`command1 ; command2`** The first command is executed, and the second one is started as soon as the first one has finished.
- **`command1 & command2`** The first command is started in the background to continue until it has finished; immediately after starting it, the second one is started as well to run in the foreground.
- **`command1 && command2`** The second command is only started if the first command is successful. To achieve this, the shell checks the exit (return) status of the first command and starts the second command only if and when that exit status is found to be "0".

- **`command1 || command2`** The second command is only started if the first command fails. The shell checks the exit status of the first command and starts the second command only if that exit status is not equal to "0".

If two commands are combined with ";" or "&", the shell will not evaluate the exit status of the first command at all. In these cases, commands are run either in sequence or in parallel, but without depending on each other.

```
tux@earth:~> w ; date
 11:27am  up 32 min,  1 user,  load average: 0.00, 0.00, 0.00
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU  WHAT
tux    :0       console          10:56am  ?      0.00s   ?     -
tux    pts/0    -                10:56am  2:34m  0.00s   ?     -
tux    pts/1    -                10:56am  0.00s  0.14s  0.01s  w
Tue Aug 21 11:27:18 CEST 2002
```

In this example, the second command is executed after the first.

Note: The shell will start a subshell for each of these two commands. If you want to redirect the output of both commands to the same destination, you can only do so by running them in the same subshell, which is done in the following fashion: `(`*`command1 ; command2`*`)`.

Example:

```
tux@earth:~> w ; date > whoandwhen
 11:27am  up 32 min,  1 user,  load average: 0.00, 0.00, 0.00
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU  WHAT
tux    :0       console          10:56am  ?      0.00s   ?     -
tux    pts/0    -                10:56am  2:34m  0.00s   ?     -
tux    pts/1    -                10:56am  0.00s  0.14s  0.01s  w
tux@earth:~>
tux@earth:~> ( w ; date ) > whoandwhen
tux@earth:~>
```

With the first sample command, only the output of `date` is directed to the file while the output of `w` still goes to standard output. But after enclosing the two commands in parentheses, they are both executed in one subshell, so their output can be redirected to the same destination.

If commands are combined with "&&" or "||", the shell evaluates the exit status of the first command to decide whether to execute the second one or not. The shell keeps the exit status of the most recent child process in the variable `?`. Thus, you can check it manually with "`echo $?`". An exit status of "0" means that the command has been run successfully. To signal a failure, the shell uses several exit status values that depend on the specific error that has occurred.

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

```
tux@earth:~> ls
file1  file3  directory1a  directory2a
file2  file4  directory1b  directory2b
tux@earth:~> echo $?
0
tux@earth:~> ls /root
ls: /root: Permission denied
tux@earth:~> echo $?
1
```

The exit status as returned by the shell will always be evaluated when using the separators "&&" and "||".

Examples:

From ~/.bashrc:

```
test -s ~/.alias && . ~/.alias
```

This checks whether the file ~/.alias exists; if it does, it is read in by the . — the short form of source.

```
tar cvzf /dev/st0 /home /etc || mail -s "Something went wrong with the backup" root
```

If an error occurs when making a backup with tar, an e-mail is sent to the system administrator to tell him about the problem.

Summary

- To change the settings for Bash on a system-wide basis, edit /etc/profile or /etc/profile.local, respectively. Users can store their personal settings in either ~/.profile or ~/.bashrc.
- Each of the three standard data channels (standard input, standard output, standard error) can be redirected to a file. By using a pipe, the output of a command can be directly fed to another command on standard input.
- With the here operator, it is possible to supply more than one line of input to commands reading from standard input (which can be used e.g., with the cat command to print several lines to the screen from within a script).
- The behavior of the shell is largely influenced by variables. A distinction must be made between shell variables (valid for the current shell only) and environment variables (inherited by any subshells).
- A distinction must be made between the shell's own built-in commands and others external to the shell. Help on built-in commands can be obtained with help *builtin-command*.

- Many characters are treated and interpreted specially by the shell. To keep the shell from interpreting these special characters as such, they need to be quoted (escaped) with each of the different quoting formats (\, "...", `...`) being suitable for different situations.

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

3 Basic Script Elements 1 (Input, Output)

In this chapter, learn

- how to create flow charts for scripts
- how to produce output from a script
- how to make scripts that read user input
- how to perform basic script operations through variables
- how to make use of arithmetic operations

3.1 Program Flow Charts

The following sections introduce and discuss some basic elements used by many programming languages. These elements are often visualized through program flow charts.

Illustrating a program through a flow chart provides the following benefits:

- they force the author to lay down the steps the script should perform to achieve the desired goal, making it clearer which constructs need to be used,
- they provide a clear symbolic outline of the algorithm, which can be used as a guide during the programming process.

These are the typical symbols used to create flow charts:

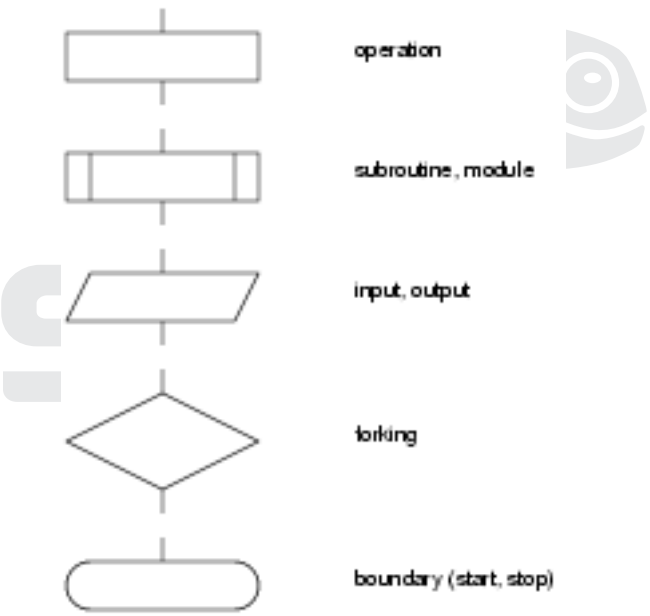


Figure 3.1: **Symbols for Program Flow Charts**

3.2 General Considerations

Before writing your first shell script, you should consider a few points about scripting in general. A shell script is basically an ASCII file containing commands that should be

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

executed in sequence. To allow this, it is important that the script is made both readable (permission symbol "r") and executable (symbol "x") for the user that will run it. However, execute permission is not granted to newly created files by default, so this needs to be done explicitly, for example, by issuing a command like `chmod +x script.sh`. Another possibility is to run the script from another shell, with a command like `sh script.sh`.

In the latter case, it is not necessary to make the script executable. Also, on the SuSE Linux Enterprise Server, `/bin/sh` is just a link to `/bin/bash`, therefore it does not really matter whether you call the script with `sh script.sh` or `bash script.sh`.

Another important point is that the directory where the script is located must actually be in the user's search path for executables. A good way to deal with this is to create a `/bin` directory for scripts under each user's home directory, such as `~/bin`.[1]

Note: If a script is not made executable but started with a command like `sh script.sh` instead, the corresponding shell will not use the normal program path. Accordingly, the script will only run if you provide the full path to the script.

When giving script files a name, it is a good idea to append the "`.sh`" suffix to it. This ensures that the file can easily be recognized as a shell script. If you do not add the suffix, you need to make sure the file name is not identical with some existing command (a common mistake is to name a script `test`).

In the section 3.3, we will discuss the basic structure of a shell script.

3.3 Producing Output From a Script

The basic structure of a shell script can be illustrated with a simple program that does nothing more than print the message "`Hello world`".

This is the flow chart for the script:

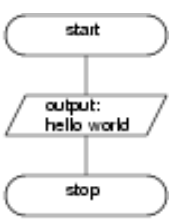


Figure 3.2: **The "Hello world" Script**

[1] On the SuSE Linux Enterprise Server, this directory is automatically added to the user's PATH variable.

So the script consists of three elements: the program start, the action, and the program stop. The action itself is the output of the words `Hello world` to the screen. The next figure shows the three elements as found in the script itself:

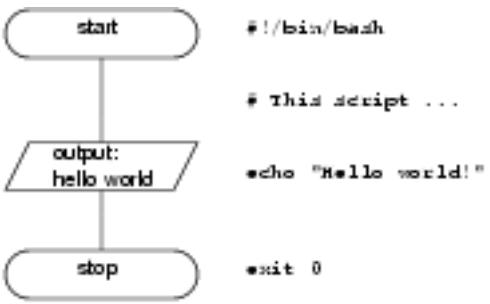


Figure 3.3: **The "Hello world" Flow Chart with Program Code**

Before having a closer look at each of the three elements, it should be mentioned that the general rules for the creation of shell scripts, as explained in this section, can be applied to any conceivable script.

1. **Start:**
   The first line of any shell script should be the so-called shebang (e.g., `#!/bin/bash`) [2]. This line specifies the shell program to call to execute the script. Just like with any other program, a subshell is started to run the script. Normally, the shell program run as a subshell will be the interactive shell from which the script is started or (in the case of non-interactive use) the user's default shell. The shebang overrides this.

   The script's start section should also include a comment describing what the script does. It is also a good idea to include the name of the author, the date, and the version number of the script. Also, any variables and functions used within the script should be defined at the top of the script.

2. **Commands:**
   Our sample script includes the `echo` command as the only one executed (to print the "Hello world" greeting). Shell scripts in general rely on the `echo` command as the most common solution to produce some (screen) output.

3. **Stop:**
   Before the script ends, it may be necessary to do some clean-up. For instance, you

[2] In the field of computing, the word "shebang" presumably owes its existence to the words "sharp" (for "#"), or maybe "shell", and "bang" (for the exclamation point) being merged into one.

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

way want to remove any temporary files created by the script. As the very last step, define the script's exit status with `exit value`, which informs the parent how the script was terminated. The exit status as returned by the script can be queried afterwards with `echo $?`.

Every script that you write should follow this basic structure.

Exercise

1. Write the "Hello world" script yourself. In the script, use the command

   ```
   echo -e "\aHello\nworld"
   ```

   to produce the output.

2. What is the purpose of the "`\a`", the "`\n`", and of the "`-e`" option?

Note: The "`\a`" will not have the intended effect in a KDE terminal window (konsole). Use an `xterm` or a `wterm`.

Solution: see Sample Script 1 in Appendix E on page 101

3.4 Reading User Input

This section describes how to write a script that accepts user input. One way to achieve this is through the `read` command. The user input is stored in a variable, which is then supplied to `read` as an argument:

```
read VARIABLE
```

The script will pause at this point, waiting for user input — until ↵ is hit. To tell the user that he is supposed to do so, we first need to print a line to that effect, which can be done with `echo`:

```
echo "Please enter a value for the variable:"
read VARIABLE
```

The following flow chart shows the general structure of a script that reads user input.

© 2004, SUSE LINUX AG (http://www.suse.com/training/)

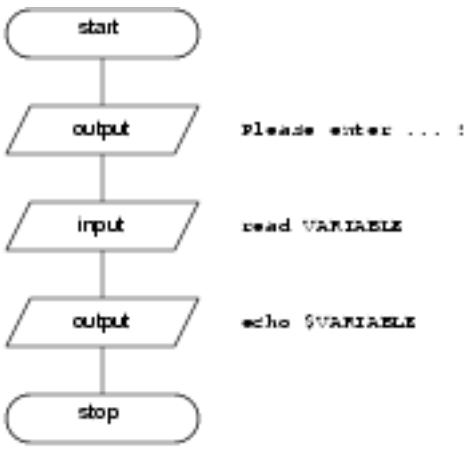


Figure 3.4: **A Simple Script Reading User Input**

First, the script produces some output with `echo ...` to ask the user to enter something. Then the `read` command waits until the input is provided to store it in a variable. In many cases, this will be followed by the script printing some output again.

Exercise

Create a simple shell script that prompts the user to enter his first and last name then greets the user with his full name.

Solution: see Sample Script 2 in Appendix E on page 101

3.5 Simple Operations with Variables

This section provides an introduction to the use of variables in shell scripts.

3.5.1 Basic String Operations

This is a simple example to show how a string value can be assigned to a variable. Again, we want to read in the user's first and last name then print both to the screen. However, this time we will create a variable called `NAME`, which holds both the first and the last name.

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

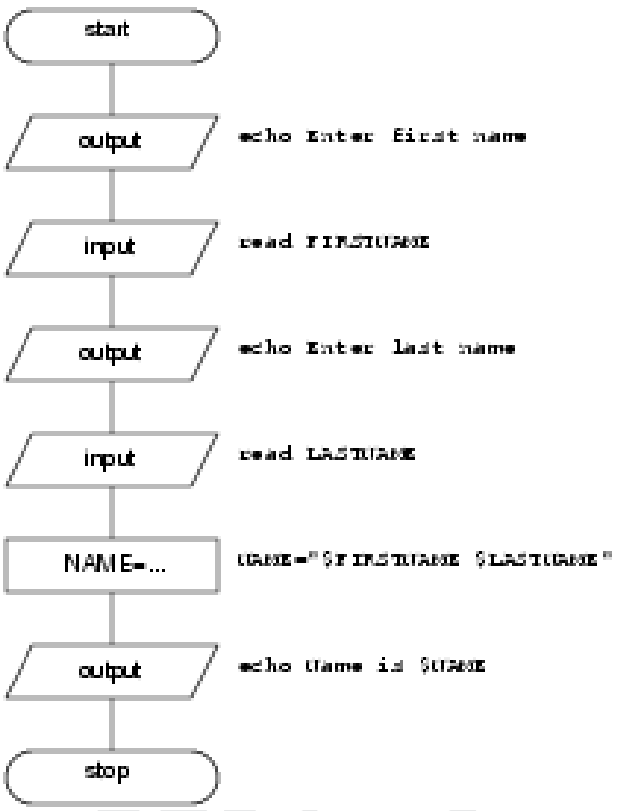


Figure 3.5: **A Script Performing a Simple String Operation**

Exercise

Write a script which implements the above flow chart.

Solution: see Sample Script 3 in Appendix E on page 102

Default Values

It is often useful to assign a default value to a variable. This may help to avoid errors, for instance, if the user has entered a value that cannot be interpreted in a meaningful way.

The following sample demonstrates how this may happen. The script prompts the user for a file name then searches for the file in the current directory.

Exercise

Create the following script:

```
#!/bin/bash
echo "Please enter the file to find:"
read FILE
# 1st variant:
find . -name $FILE
# 2nd variant:
# find . -name "$FILE"
```

Now test either variant of the script. How do the two variants behave if the user just presses ⏎ instead of entering a file name?

Bash lets you use special variable substitution patterns (see Appendix C on page 97) to forestall such problems by setting a default value for a variable. The default is used whenever the variable is left empty or is not assigned at all. In the case of our sample script, the default can be set in the following way:

```
${FILE:="*.bak"}
```

If the user does not specify any file name, the script searches for all files whose names are matched by `*.bak`.

Exercise

Extend your script according to the above sample line.

Solution: see Sample Script 4 in Appendix E on page 102

3.5.2 Arithmetic Operations

It is often desirable in shell scripts to use values assigned to variables for some kind of calculation. There are several possibilities to implement this.

The Bourne shell is limited in this regard, but can perform such operations by relying on external commands (such as `expr`).

The Bash shell comes with built-in support for arithmetic operations, but there are some limitations to this as well. Specifically, the arithmetic capabilities of Bash are limited in the following way:

- Only operations with whole numbers (integers) can be performed.
- All numbers are 32-bit values with no overflow check.

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

So even when using Bash, you may need to use external commands, for instance `bc` for floating-point calculations.

The following paragraphs list all the possible methods and formats for arithmetic operations. All of them implement this sample operation:

Example: **A=B+10**

- Using the external command `expr` (Bourne shell compatible):

  ```
  A=‘expr $B + 10‘
  ```

  Given that an external command is used, this method will also work with the Bourne shell. It should be remembered, however, that scripts using external commands will always perform slower than those relying on built-in commands.

- Using the Bash built-in `let` :

  ```
  let A="$B + 10"
  ```

  In Bash, the `let` command can be used to introduce an arithmetic expression.

- Arithmetic expressions inside parentheses or brackets (two different formats) (see Section 2.8.4 on page 28):

  ```
  A=$((B + 10))

  A=$[B + 10]
  ```

  Arithmetic expressions can be enclosed in double parentheses or in brackets for expansion by Bash. Both `$((...))` and `$[...]` are possible, but the latter is considered deprecated and should be avoided.

- Using the built-in `declare` to declare a variable as an integer :

  ```
  declare -i A
  declare -i B

  A=B+10
  ```

  If all the variables involved in a calculation have previously been declared as integers through `declare-i`, arithmetic evaluation of these variables happens automatically when they are assigned a value. This means that the variable `B`, for instance, does not have to be prefixed with the "$" to be evaluated.

With the `expr` command, only the following five operators are available: "+", "-", "*", "/", and "%". Additional operators (which are identical to those of the C programming language) can be used with all of the above Bash formats. For a complete list, consult the Bash manual page.

Of course, it makes sense to limit oneself to just one of these possibilities. As far as Bash is concerned, a good choice may be to only use the `declare` command, as it makes the best use of the available features.

Exercise

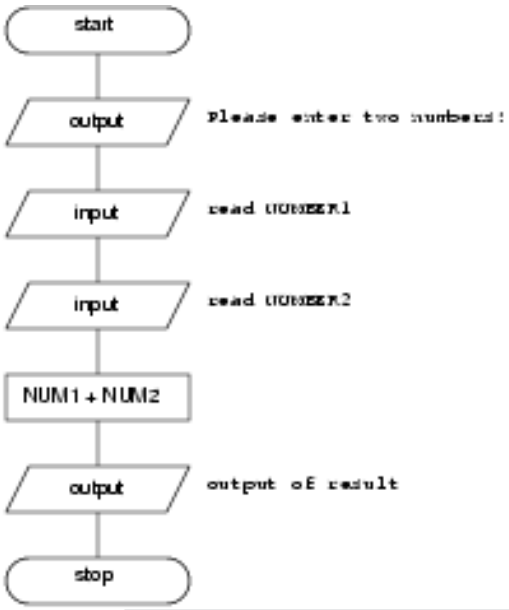


Figure 3.6: **Performing Arithmetic Operations under Bash**

1. Implement a script on the basis of the flow chart. The script should make use of all the formats in which arithmetic operations are possible under Bash.

   Solution: see Sample Script 5 in Appendix E on page 103

2. Modify the script to use the other fundamental arithmetic operations (-, *, /).
3. What happens if the user
   - enters a word for each number?
   - enters nothing at all, but only hits ⏎ at each prompt?

Summary

- Before writing a shell script, it is useful to draw a symbolic outline in the form of a program flow chart.
- Before a file can be run as a shell script, it must have both read and execute permission.

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

- To produce some simple output from a script, you can use the `echo` command.
- To read user input for processing by a script, you can use the `read` command.
- There are several ways to perform arithmetic operations in a script. One of them is the external command `expr`. Another one is to enclose arithmetic expressions in double parentheses for expansion by the shell. In Bash, arithmetic operations can also be performed with plain variables, provided that these have been declared as integers before.

© 2004, SUSE LINUX AG (http://www.suse.com/training/)

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

4 Basic Script Elements 2 (Control Structures)

In this chapter, learn

- how to create basic branches with `if`
- how to build multiple branches with `case`
- how to create loops using `while` and `until`
- how to process lists with `for`
- how to interrupt loop processing

4.1 Simple Branching With `if`

With the following structure, create a simple branch within a script:

```
if condition
then
   commands
fi
```

The `if` statement can be extended with an optional `else` in this way:

```
if condition
then
   command1
else
   command2
fi
```

In a program flow chart, a branch created with an `if` statement can be represented like this:

Figure 4.1: **Branching with `if`**

A branch of this type must begin with `if` and end with `fi`. The exit status of the first command decides whether the command introduced by `then` will be executed — the latter will be executed only if the exit status is zero (meaning that the previous command has been run successfully without error). If the exit status is not zero, the shell goes to the end of the branch or, if present, to `else`.

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

When using these control structures in a shell script, individual commands, such as `if`, `then`, and `fi`, must follow immediately after a command separator. In the above case, the separator is a new line. It could also be a semicolon, which would allow us to enter the same `if` statement as one command line:

`if` *`condition`*`; then` *`commands`*`; fi`

The following paragraphs use a sample script to explain how an `if` branch works in practical terms. This sample script asks the user to enter his date of birth; if that happens to be today, the scripts congratulates him on his birthday — but does nothing special if the day is another one.

Figure 4.2: **Sample Script with an `if` Branch**

There are a number of points to be taken into account when writing this script. From the flow chart, it should be obvious that the script basically consists of two steps:

1. Prompting the user to enter the date of birth.
2. Comparing the date as entered by the user with the current date. If the dates are the same, the users receives our congratulations. If they are not equal, there are no congratulations.

The branch is the actual mechanism by which the current date and the date of birth are compared. Before the comparison can go ahead, both dates must be available in the same format. Also, the user should be asked to specify the date of birth in a suitable format.

First and foremost, we need to know the format in which the current date can be obtained from the system. The obvious choice to get a date string is `date` , which can be run with the `-I` option[1] to suppress any information about the current time:

```
tux@earth:~> date -I
2002-09-09
```

Now that we have determined the date format, we make sure that the user enters the date in exactly the same way:

```
echo "Please enter your date of birth (YYYY-MM-DD, for instance, 1973-12-21): "

read BIRTHDAY
```

As far as the second step is concerned, it turns out that there are several things involved in it. To determine whether today is the user's birthday, we need to compare the two dates. The user's date of birth is already stored in the `BIRTHDAY` variable, but we still need to create another variable to store the current date. This can be done through command substitution:

```
TODAY=‘date -I‘
```

At this point it should become clear very soon that the mere comparison of the two dates as stored in the variables `BIRTHDAY`, 1973-12-21, and `TODAY`, 2002-09-09) will not achieve what we want. To determine the birthday, we first need to do away with the years then compare the remaining date strings only. To achieve this, we can use one of the variable substitution mechanisms of Bash (see Appendix C on page 97), which in this case cuts out the year part and returns the rest. So in its final form, the first part of the script should look like this:

```
#!/bin/bash
echo "Please enter your date of birth (YYYY-MM-DD, for instance 1973-12-21): "

read BIRTHDAY

BIRTHDAY=${BIRTHDAY#[0-9]*-}

TODAY=‘date -I‘

TODAY=${TODAY#[0-9]*-}
```

Now we are able to compare the two values with the help of an `if` branch. Variables are mostly compared using the `test` command (see Appendix A.7 on page 91).[2] The `test` command is followed by a string condition such as `test $VARIABLE1 = $VARIABLE2`. If the condition is met (if the value of `VARIABLE1` is identical to the value of `VARIABLE2`), `test` returns a zero to indicate success.

[1] Actually, the `-I` option tells `date` to output the date in ISO format (see Appendix A.3 on page 86).
[2] Instead of the regular `test ...`, you can also use the short form `[...]`.

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

So the second part of the shell script could look like this:

```
if      test "$BIRTHDAY" = "$TODAY"
then
        echo "Tada! Happy birthday to you! Nice presents awaiting you ..."
else
        echo "Sorry to disappoint you, no presents today ..."
fi
```

As its last action, we want the script to use the `exit` command to finish with a certain exit status, which depends on whether today is the user's birthday or not. This is implemented by defining yet another variable (for the complete script, see E on page 104):

```
if      test "$BIRTHDAY" = "$TODAY"
then
        echo "Tada! Happy birthday to you! Nice presents awaiting you ..."
        STATUS=0
else
        echo "Sorry to disappoint you, no presents today ..."
        STATUS=1
fi

exit $STATUS
```

This sample script is just one example of how you can use an `if` statement whenever it is necessary to compare two values with the next action depending upon the result of this comparison.

4.1.1 Short Forms of `if`

In Section 2.9.1 on page 31, we discussed various ways of executing several commands in sequence. This included the separators "`&&`" and "`||`", which make it possible to execute a second command depending on the success or failure of the first:

`command1 && command2`
`command1 || command2`

Now these separators can also be understood as short forms of an `if` branch.

In both cases the shell checks the exit status of the first command to decide whether the second command is run.

Therefore, the structure

```
if      test -e file
then
        . file
fi
```

can be condensed into the following command line:

```
test -e file && . file
```

Whenever the comparison is a simple one as in the above example, you can replace the relatively complex `if - then - fi` structure with a command line that uses "&&" or "||" to chain the commands.

Exercises

Exercise 1

1. Write a shell script that checks whether a given file exists. The script shall inform the user about the result. The exit status of the script shall depend on this result — on the existence of the file. The name of the file shall **not** be read with the `read` command, but the script should be able to accept it as an argument (as in `file.sh filename`).

   Note: The script should accept only one file name as an argument, so it needs to check how many arguments are passed on to it (see Section 2.3.2 on page 12). The script shall check for the existence of the file only after having checked the user input in this way.

   Solution: see Sample Script 7 in Appendix E on page 105

2. The script shall output an error message if the user input is not correct. Assuming that this message is produced with `echo`, it will be printed on standard output by default, but you should enhance the script such that the message goes to standard error. Also, make sure that the message explains which kind of user input is actually expected.

   Solution: see Sample Script 8 in Appendix E on page 107

Exercise 2

Write a script to check the disk space being used. The script should print a warning if 90 percent or more of the disk space is used on one of the mounted partitions.

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

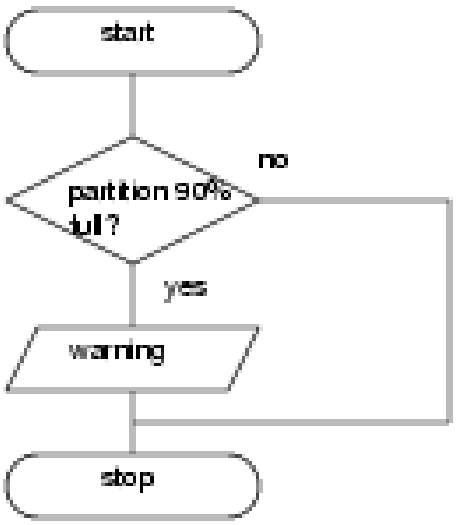


Figure 4.3: **Checking the Amount of Space Used by Partitions**

Note: To check to amount of space used by a file system, you can use the `df` command. The recommended tool to filter out the percent value is the `egrep` command.

Solution: see Sample Script 9 in Appendix E on page 108

4.2 Multiple Branches With case

Apart from simple branches with `if`, it is also possible to create multiple branches with `case`. In a `case` statement, the expression contained in a variable is compared with a number of expressions and for each expression matched a command is executed.

A `case` statement has the following structure:

```
case $VAR in
   expression1) command1;;
   expression2) command2;;
…
esac
```

In a flow chart, multiple branching with `case` looks quite similar to a simple branch created with `if`:

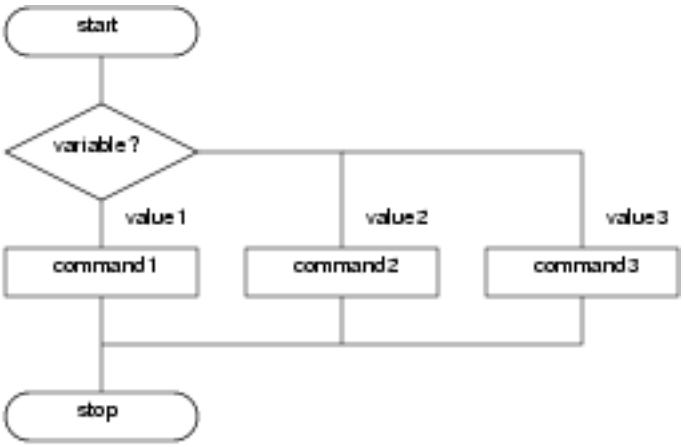


Figure 4.4: **Multiple Branching with `case`**

To better understand how a multiple branch works, we will discuss the details of a sample script. This script prompts the user to enter the name of an animal. The name is then stored in a variable and compared with a number of possible matches. For the matches found, the script will tell the user how many legs the given animal has. To allow for several expressions to be matched within one and the same branch, several expressions can be listed on one line with a "`|`" as a separator. The script might then look like this:

```
#!/bin/bash

cat << EOF
Name me an animal and I will tell
you how many legs it has!
EOF

read CREATURE

case    "$CREATURE" in
        dog | cat | mouse )
                echo "A $CREATURE has 4 legs."
                ;;

        bird | human | monkey )
                echo "A $CREATURE has 2 legs."
                ;;
...
```

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

```
        spider )
             echo "A $CREATURE has 8 legs."
             ;;

        fly )
             echo "A $CREATURE has 6 legs."
             ;;

        * )
             echo "I haven't the faintest idea how many legs a(n) $CREATURE has."
             ;;
esac

exit 0
```

As you can see, the user input is read in and assigned to the CREATURE variable. The case statement then compares this value against each of the expressions provided as alternatives. For instance, if the user enters "cat" as the animal, the script prints the matching sentence that says that this animal has four legs. The asterisk is often used as the last expression to be evaluated to cover all cases not matched by the previous alternatives. The corresponding message states that the number of legs is not known.

It is important to make sure that the expressions provide for an exact match of any allowable expression. For instance, if someone entered "Dog" instead of "dog", the script will pretend that it does not know the number of legs for this strange kind of animal. Therefore, it is useful to supply the possible alternatives beforehand:

```
...
case    "$CREATURE" in
        [dD]og | [cC]at | [mM]ouse )
...
```

You can provide such alternatives in brackets, in the same way as with the shell's filename expansion mechanism discussed before (for the complete script, see Appendix E on page 109).

Exercise

Write a script section (not a complete script) to make a proposal of a how script could use a case statement to process the user's answer to a yes/no question.

Solution: see Sample Script 11 in Appendix E on page 110

4.3 Iterations and Loops

4.3.1 Looping With `while` and `until`

The purpose of a loop is to test a certain condition and to execute a given command "while" the condition is true (`while` loop) or "until" the condition becomes true (`until` loop).

`while` *condition*	`until` *condition*
`do`	`do`
commands	*commands*
`done`	`done`

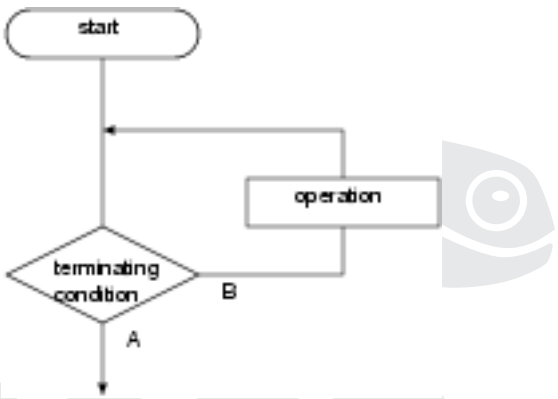


Figure 4.5: **Structure of a `while` or `until` Loop**

These loops actually rely on the exit status of a terminating condition: a `while` loop remains operative as long as the condition's exit status is zero (path "B" in the flow chart), but an `until` loop is terminated if the status is zero (path "A" in the flow chart). By contrast, a `while` loop is terminated when the exit status becomes non-zero (when the condition is not true), but an `until` loop is operative as long as the status is non-zero.

Exercise

1. Write a script in which a simple `while` loop is performed 100 times.

   Solution: see Sample Script 12 in Appendix E on page 111

2. Write another script in which an `until` loop is performed 100 times.

   Solution: see Sample Script 13 in Appendix E on page 112

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

4.3.2 Processing a List With `for`

The purpose of a `for` loop is to process a list of elements. It has the following syntax:

```
for variable in element1 element2 element3
do
    commands
done
```

A `for` loop executes the given commands once for every element of the list and the value of the variable matches one list element with each loop iteration. The list itself is often created through command substitution.

As with similar constructs, a command separator must immediately precede the "`do`" and "`done`" parts of the `for` loop. Accordingly, a `for` loop can be entered in two different ways:

```
tux@earth:~> for i in 1 2 3 4 5 6 7 8
> do ping -c1 earth$i
> done
```

In the above case, the command separator is a line break and the loop is started only after entering the final "`done`". One could instead use a semicolon as a separator, in which case the same loop takes the form of:

```
tux@earth:~> for i in 1 2 3 4 5 6 7 8; do ping -c1 earth$i; done
```

Exercise

Write a shell script that renames all files in the current directory, such that uppercase letters in file names are transformed to lowercase.

Note: You can use the command `tr` to replace characters in file names. To avoid mistakes, however, you should not rename files without any previous testing. If you put the `echo` command before the actual renaming command, the script will tell you which characters would be replaced in which file names without performing the real action.

Solution: see Sample Script 14 in Appendix E on page 113

4.4 Exiting From a Loop

4.4.1 Exiting From the Current Loop Iteration with `continue`

The `continue` command makes it possible to exit from the current iteration of a loop (`while`, `until`, `for`, and `select`) to resume with the next iteration of the loop. By do-

ing so, a script can test for an additional condition with each iteration without affecting the loop as a whole — without stopping it altogether (as a result of the terminating condition becoming true, for instance).

Example:

```
for     FILE in ‘ls *.mp3‘
do
        if      test -e /MP3/$FILE
        then
                echo "The file $FILE exists."
                continue
        fi

        cp $FILE /MP3
done
```

This script writes a backup copy of all files ending with "`.mp3`" to the directory `MP3` unless there is already a file with the same name in that directory. In the latter case, the script prints a message stating that the file already exists and exits from the current loop iteration.

In the case of loops nested within each other, a number can be supplied to the `continue` command as an argument to exit from the current iteration and resume with the `n`th next upper loop.

Exercise 1

Enhance the script written in the exercise of Section 4.3.2 on the preceding page to include an additional condition, such that no existing files can be overwritten. To implement this, use `continue` to exit from the current loop iteration if renaming the file would overwrite another one.

Solution: see Sample Script 15 in Appendix E on page 114

Exercise 2

Expand the script from exercise 1 in such as way that the loop is not terminated but continues to query a new file name until a file name is entered that does not yet exist.

Solution: see Sample Script 16 in Appendix E on page 115

4.4.2 Exiting from a Loop with `break`

The previous section discussed how `continue` can be used to exit from the current iteration of a loop. The `break` command is another way to introduce a new condition within

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

a loop. Unlike `continue`, however, it causes the loop to be terminated altogether if the condition is met (and not only the current loop iteration).

Summary

- Conditional statements in shell scripts can be implemented with an `if` branch. For relatively simple structures, you can also use the command separators "`&&`" and "`||`" to express the same statement as a command line.
- To take decisions with a number of possible choices in a script, create a multiple branch with a `case` statement.
- With the commands `while` and `until`, create loops that depend on certain terminating conditions.
- The `for` command allows you to create loops to process a list of elements.
- There are two possibilities to influence the operation of a loop: With the `break` command, a loop can be terminated altogether according to a given condition. By contrast, the `continue` command allows exiting from the current iteration of a loop if the condition is true.

© 2004, SUSE LINUX AG (http://www.suse.com/training/)

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

5 Advanced Scripting Techniques

In this chapter, learn

- how to use `read` for more complex user input
- how to define shell functions
- how to read in options for a script
- how to implement selection menus with `select`
- how to create dialogs using the `dialog` package as an add-on.

5.1 Reading Input With `read`

Section 3.4 on page 39 discussed how `read` can be used to read in a simple user-supplied string or number:

Example:

```
echo "Please enter a number: "
read NUMBER
```

At this point, the script will pause and wait for the input completed with ⏎. The string entered is then assigned to the `NUMBER` variable.

However, `read` is also able to read in several lines of input, for instance, from a file or from a command's standard output. This can be done with the following syntax:

commands | while read VAR1 VAR2
do
  commands
done
	while read VAR1 VAR2
	do
	  commands
	done < *file*

With the help of a `while` loop, `read` processes the input line by line as available from the output of a command (left column, redirection through a pipe) or from a file (right column, redirection of stdin with "<"). The strings read in are assigned as values to the specified variables. Accordingly, the number of variables specified should correspond to the number of strings expected. A construct of this type makes it easy to read file contents or command output line by line to process them in a step-by-step fashion.

Example:

```
IFS=:
cat /etc/passwd |  while read LOGIN PASS USERID GROUP REST
do
...
```

This processes the file `/etc/passwd`, reading its contents line by line. However, given that the entries on each line as present in this file are not separated by spaces but colons, we first need to change the internal field separator by setting the `IFS` variable to a different value. This ensures that each entry in a line is assigned as a value to a variable. The `while` loop performs one iteration per input line, assigning the first entry of the `passwd` file to the `LOGIN` variable, the second entry to the `PASS` value, and so on. Given that only five variables have been defined even though each line has seven entries, all the remaining entries are assigned to the last variable — entries five to seven are assigned to the `REST` variable.

Exercise

Modify Sample Script 9 to be more flexible. In its original version, the script just printed a warning message if 90 percent or more of the available space was

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

used on one of the partitions. Rewrite the script to accept a warning threshold as an argument (*scriptname threshold*, for example, `df2.sh 85`) but with "90%" still being used as the default limit if no such argument is supplied by the user. Furthermore, rewrite the warning message to be more specific, so it indicates how much space is used on which partition.

Solution: see Sample Script 17 in Appendix E on page 116

5.2 Shell Functions

Shell functions could also be called script modules, because they make a whole script section available under a single name. Shell functions are normally defined at the beginning of a script. Alternatively, you can store several functions in a file read whenever the functions are needed.

The syntax to define a function is as follows:

```
functionname () {
    commands
    commands
}
```

The function name may be composed of any regular character string that then can be used to call the function.

For those functions that may be useful in many different scripts, it is a good idea to store them all in one file, such as ~`/bin/useful-functions`. These are some practical examples:

```
# mcd: mkdir + cd; creates a new directory and
# changes into that new directory right away

mcd (){
        mkdir $1
        cd $1
}
```

After having been created, this function can be called as `mcd` *`dirname`*. The shell will create the specified directory and immediately change into that directory.

```
# pause: causes a script to take a break

pause (){
        echo "To continue, hit RETURN."
        read q
}
```

This function can be used to create a pause in a script. The script will only resume after ⏎ is pressed.

Functions can also be written such that their processing is stopped from within, in a way very similar to exiting a loop (iteration) with the commands `break` and `continue` (see Section 4.4.1 on page 57). The basic difference is that within a function, the command `return` is needed to exit from it. If `return` is called without an argument, the return value of the function is identical to the exit status of the last command executed in that function. Otherwise, the return value is identical to the one supplied as an argument to `return`.

Example:

```
funkyfunct (){
        command1

        if      ...
        then
                return 0
        fi

        command2
}
```

The following function lets a user answer a yes-or-no question. It uses the `case` construct as introduced with Exercise 4.2 on page 55 and Sample Script 11 in Appendix E on page 110:

```
# Prompt the user to answer with "yes" or "no".
# The question itself is supplied as an argument
# when calling the function, e.g.:
# "yesno Do you want to continue?"

yesno (){
        while   true
        do
                echo "$*"
                echo "Please answer by entering (y)es or (n)o:"
                read ANSWER
                case "$ANSWER" in
                        [yY] | [yY][eE][sS] )
                                return 0
                                ;;
                        [nN] | [nN][oO] )
                                return 1
                                ;;
                        * )
                                echo "I can't understand you over here."
                                ;;
                esac
        done
}
```

When the function is called, a question is supplied to it as an argument, like this: "`yesno Do you want to continue?`". If you remember that all arguments passed to commands are stored in variables (see Section 2.3.2 on page 12), it should be clear how the question passed to the function can be extracted with `echo "$*"`. Further,

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

if the user answers with "yes", the function exits with a return value of "0". If he enters "no" instead, the return value is "1". The combination of `while` and `true` constitutes an infinite loop, so the question is repeated again and again until the user enters a valid answer, causing the function to exit via `return`.

Example:

```
...
yesno Do you want to continue? || exit 1
...
```

Here, again, the user is prompted to answer the question "`Do you want to continue?`". If the answer is "yes", the script will continue. If it is "no", the function's return value is "1", so `exit 1` is executed, causing the script to terminate.

Exercises

Exercise 1

Use the above `yesno` function to write a script that lets the system administrator delete user accounts. The script should prompt for the account to delete then ask whether the user's home directory should be deleted as well. If the latter question is answered with "no", the script shall change the user and group ownership of the corresponding home directory to *root*. After doing so, the script shall use the `yesno` function again to ask whether the administrator really wants to delete the account.

Note: Before running the script for the first time, you should put an `echo` before the commands executed by the script (`chown`, `userdel`). By doing so, you can also avoid creating a new test account for each new run of the script.

Solution: see Sample Script 18 in Appendix E on page 117

Exercise 2

Enhance the script written in Exercise 5.1 on page 62 (Sample Script 17 in Appendix E on page 116) to include a help function, so a help text is printed if the user input was incorrect.

Solution: see Sample Script 19 in Appendix E on page 118

© 2004, SUSE LINUX AG (http://www.suse.com/training/)

5.3 Reading Options With `getopts`

With the shell built-in command `getopts`, you can extract the options supplied to a script on the command line. The shell interprets command-line arguments as command options only if they are prefixed with a "-" (the default when using the shell interactively). This makes it possible to place options in different positions on the command line and to supply them in an arbitrary order. Therefore, the command

```
tux@earth:~> cp -dpR *.txt texts/
```

achieves the same thing as the command

```
tux@earth:~> cp  -R *.txt -d texts/ -p
```

The `getopts` shell built-in recognizes options in the same way. The command requires the following syntax:

```
getopts optionstring VAR
```

The *`optionstring`* describes all options to be recognized. For instance, `getopts abc` declares "a", "b", and "c" as the options to be processed. If a parameter is expected for the option (for instance, `-m` *`maxvalue`*), the corresponding string must be followed by a ":" (as in `getopts m:`). The option string is followed by a variable to which all the command-line options specified are assigned as a list. The `getopts` command is mostly used in a `while` loop together with `case` to define which command to execute for a given option:

```
while getopts optionstring VAR; do
   case $VAR in
      optionstring1 ) command1 ;;
      optionstring2 ) command2 ;;
   esac
done
```

Exercise

Enhance the script of Exercise 5.2 on the page before (Sample Script 19 in Appendix E on page 118) such that it can accept options from the command line. Also, the script shall not read the threshold as a direct numeric argument anymore, but through a command-line option. As in the previous version of the script, the default threshold shall be 90 percent. The script shall be able to accept all of the following options:

`-h` Show the help text.

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

`-m` Do not print the message about how much space is used on which partition to standard output, but mail the message to *root* instead.

`-t` ***`threshold-percentage`*** Set the warning threshold to the given percentage (from 1 through 100).

Solution: see Sample Script 20 in Appendix E on page 120

5.4 Signal Handling With `trap`

Signals can be sent to any running process on the system with the `kill` command. It normally takes the form of `kill -SIGNAL` *`PID`* where `SIGNAL` specifies the signal name or number (see below) and `PID` the process ID of the corresponding process. For foreground processes running in a terminal, it is also possible to send some of the signals directly via key combos. The signals available on the system can be listed with `kill -l` and an explanation of the action taken by each signal can be obtained with `man 7 signal`. However, not every program understands every signal. With the exception of number 9 (SIGKILL), the Linux kernel sends all signals to the given process, after which it depends on the program's capabilities whether the signal is respected. Signal 9 is directly handled by the kernel and is therefore applicable to all programs (the only exception being `init`).

Among the more common signals, these are the ones with which you should be familiar:

Signal 1 — SIGHUP: Most shells send this signal to all processes started from it when the shell is terminated. The signal causes the corresponding processes to exit.

Signal 2 — SIGINT: This signal can be sent to a process running as a foreground task in a terminal by pressing Ctrl + C. The signal requests the process to exit, but allows it to do so in an "orderly" manner — any buffer data can be written to disk before exiting.

Signal 9 — SIGKILL: Like signal 19 (SIGSTOP), this signal is not handled by the process but directly by the kernel. It causes the process to abort immediately with any buffer data being discarded.

Signal 15 — SIGTERM: This is the default signal sent by the `kill` command — the one sent when `kill` is run without specifying any signal. It is similar to signal 2 and requests the process to exit.

Signal 18 — SIGCONT: This signal causes a process with the status "T" (stopped or traced) to continue. It is also used to resume a process started interactively with the built-ins `fg` and `bg`.

Signal 19 — SIGSTOP: This signal can be sent to a process running as a foreground task in a terminal by pressing Ctrl + Z. It causes the process to be suspended giving it the status "T". The process can be resumed later by sending signal 18 to it.

Signals that cannot be generated through a key combination must be sent with the `kill` command. When calling `kill` without any further options (`kill` *PID*), it sends the default signal 15 (SIGTERM). To send a particular signal to a process, you can specify it as an option for `kill` either in the form of a number (e.g., `kill -9`) or by writing the name of the signal (e.g., `kill -SIGKILL` or the short form `kill -KILL`).

Shell scripts may behave in an undesirable way if they are abruptly stopped, for instance, by hitting Ctrl + C. If that happens, the script might leave behind a number of temporary files. The `trap` command helps to avoid this by intercepting signals to compensate for their action by running certain commands. The `trap` command requires the following syntax:

```
trap command signal
```

For example, the line `trap` *command* `2 15` would cause a script to intercept signals 2 (SIGINT) and 15 (SIGTERM) and run the specified command instead. The `trap` built-in can also be run without specifying a command (as in `trap "" HUP`), in which case the signal will simply be ignored.

In general, signal handling should be defined at the beginning of a script. As soon as one of the declared signals is intercepted, `trap` will take care of running the corresponding command for it.

Note: The `trap` built-in can intercept all signals in this way except signal 9 (SIGINT) and signal 19 (SIGSTOP), because the kernel has exclusive command over this signal.

Exercise

Modify Sample Script 20 (Appendix E on page 120) to properly handle signals 2 and 15. If either signal is intercepted, the script should first delete the temporary file then terminate with `exit`, so the script exits with a unique exit status.

Solution: see Sample Script 21 in Appendix E on page 122

5.5 Implementing Simple Menus with `select`

With Bash, it is possible to create simple menus with the help of the `select` built-in command.

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

This is the syntax of select:

```
PS3=prompting-text
select VARIABLE in item1 item2 item3
do
    commands
done
```

The actual text that prompts for the action (such as Please select a file) is assigned to the tertiary prompt variable "PS3". When the menu is displayed, this text appears below the actual items, each of which has a number. The user is supposed to reply by entering one of the numbers that correspond to an item. The item's contents are then stored in the variable. In most cases, a case construct is used within the select loop to define a command for each item:

```
PS3=prompting text
select VARIABLE in item1 item2 item3
do
    case VARIABLE in
        value1 ) command1 ;;
        value2 ) command2 ;;
    esac
done
```

If the user does not select any of the items but simply hits ⏎, the loop will resume and display the menu again. Just like with a for, a while, or an until loop, a select loop allows you to exit from the current iteration with continue or from the loop as a whole with break (see Section 4.4.2 on page 58).

Here is an example to show how select can be used to create a basic menu:

```
#!/bin/bash
# Creates a simple menu using select

PS3="Please select one of the items: "
select VAR in Item1 Item2 Quit; do
        case    "$VAR" in
        Quit )
                echo "You chose to quit me."
                break
                ;;
        * )
                echo "You've selected $VAR."
                ;;
        esac
done
```

When executing the script, its output should look like this:

```
tux@earth:~> bin/select-test
1) Item1
2) Item2
3) Quit
Please select one of the items: 1
You've selected Item1.
1) Item1
2) Item2
3) Quit
Please select one of the items: 3
You chose to quit me.
```

The next example shows how `select` can be used in a script that analyzes the system's passwords. The script informs about accounts with a usablepassword, with no usable password (account is locked), or with no password. Only *root* is allowed to view this information, which is implemented in the script by checking whether it was actually called by *root*.

```
#!/bin/bash
# This script lets you see which users have
#       - a usable password
#       - no usable password (account is locked)
#       - no password
# Based on "passwd -S", which displays the account status
# if the account name is supplied as a parameter.

# exit from this script if not called by root
if [ "$UID" -ne 0 ]
then
        echo "Sorry, you must run this script as root. Account \
        status information is not available for regular users."
        exit 0
fi

# define a function to check the account status for all users
# listed in /etc/passwd
get_account_status () {
        for     ACCOUNT in ‘grep ^[a-z] /etc/passwd | cut -d : -f 1‘
        do
                PASSW=‘passwd -S $ACCOUNT | cut -d " " -f 2‘

                if      test "$PASSW" = "$STATUS"
                then
                        echo $ACCOUNT
                fi
        done
}

cat << THE-END
This script checks the password status of user accounts.
According to the selection made, it lists all accounts
for which there is a usable password, no usable
password (account is disabled), or no password.
THE-END
```

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

```
PS3="Please select one of the above:"
select i in "Usable password" "Account locked" "No password" Quit
do
        if [ "$i" = "Quit" ]
        then
                break
        else    case $i in
                        "Usable password" )
                                echo "Accounts with a usable password:"
                                STATUS="PS"
                                get_account_status
                                ;;
                        "Account locked" )
                                echo "Accounts that are locked (disabled):"
                                STATUS="LK"
                                get_account_status
                                ;;
                        "No password" )
                                echo "Accounts that have no password:"
                                STATUS="NP"
                                get_account_status
                                ;;
                esac
        fi
done
```

After printing some information to explain what it does, the script uses `select` to present a menu from which the user may choose among the available actions. A `for` loop is used to get the user names in `/etc/passwd` and to filter out a string identifying the status (such as "NP" for no password) from the output of `passwd -S`. The final output lists all user names with the status selected from the menu.

Exercise

Write a script that accepts a group name as an argument. The script should present a menu to let the user decide whether to list: 1) all users of the given group, 2) the users who are primary members of the group, 3) the users who are secondary members of the group.

Note: Primary group membership can be obtained from the file `/etc/passwd`, but to find out about secondary group members, you will have to scan `/etc/group`. Therefore, the script will have to search only one of these files or both of them depending on the user's selection in the menu. It will be helpful to declare a number of functions for the script, which are then called according to the selection made.

Solution: see Sample Script 22 in Appendix E on page 124

5.6 Dialog Boxes With `dialog`

Before dialogs can be created from a shell script, you need to install `dialog` as an additional package.

The `dialog` package allows you to present different types of boxes, each for a different task. For instance, you can create menus to let the user select an item, boxes with entry fields to read user input, or lists to allow enabling or disabling certain items. The user can cycle through the interface elements with Tab and use the space bar to select an element.

If you are using the KDE desktop, it is also possible to create "real" graphical user interface elements with `kdialog`, which actually relies on the same syntax as `dialog`. However, not all of the latter's options are supported by `kdialog`. However, it should be quite easy to adapt scripts originally written for `dialog`. [1]

The `dialog` command requires this syntax:

```
dialog --boxtype "text" height width
```

As can be seen, the height and width of the box (number of rows and columns) must always be specified together with the type of box (see below). Some box types require additional parameters.

The following sections provide an overview of the most frequently used box types that `dialog` can display.

5.6.1 Yes/No Box (`yesno`)

Figure 5.1: **A `yesno` Box**

[1] The most important difference is that `dialog` directs all user input to standard error, while `kdialog` directs it to standard output.

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

A box of this type displays some text and two button-like widgets below it for selecting yes or no. The exit status returned by dialog depends on the user's selection — a status of "0" is returned if yes is selected and "1" if no is selected.

Example (see Figure 5.1 on the facing page):

```
dialog --yesno "Would you like to continue?" 10 50
```

Optionally, define a title for the box with --title, like in this example:

```
dialog --title "yesno box" --yesno "Would you like to continue?" 10 50
```

The title appears at the top border of the box:

Figure 5.2: **Dialog Box with Title**

The purpose of a yesno box is to prompt the user for an answer (yes or no) and to make the input available through the exit status of dialog ("0" or "1", respectively), which can then be queried by a shell script.

5.6.2 Message Box (`msgbox`)

Figure 5.3: **A `msgbox`**

This type of box allows you to display some text to be read by the user, who is supposed to confirm the message by selecting the `OK` button.

Example (see Figure 5.3):

```
dialog --msgbox "This is a message." 10 50
```

If the message comprises several lines of text, `dialog` will take care of the necessary line breaks automatically, according to the box width specified (50 characters in our example). However, you can also insert a fixed line break manually with a "`\n`".

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

5.6.3 Input Box (inputbox)

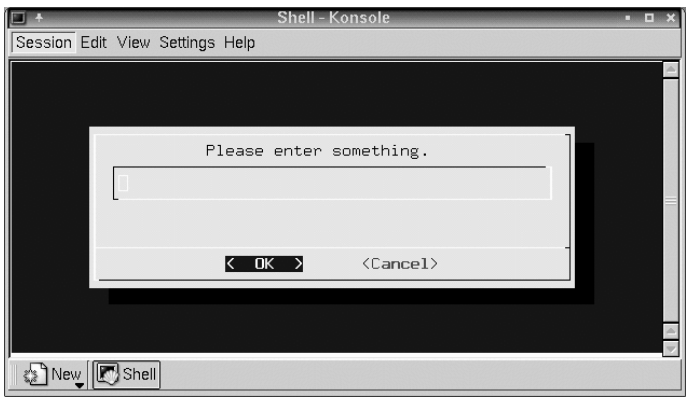


Figure 5.4: **Prompting for Input with an inputbox**

With this type of dialog box, prompt the user to supply some kind of input. The input strings, however, cannot be processed directly, but must always be saved to a temporary file. This is due to the fact that dialog uses standard output to display its boxes so needs to direct all user input to standard error. Therefore, the latter is stored in a temporary file, after which it can be assigned to a variable for further use.

This example shows how the procedure works:

```
dialog --inputbox "Please enter something." 10 50 \
2> /tmp/tempfile
VAR=‘cat /tmp/tempfile‘
```

As can be seen, standard error is redirected to /tmp/tempfile. In the next step, the contents of this file are assigned to VAR through command substitution. The input box created by this command sequence is shown in Figure 5.4.

Dialog boxes of the inputbox type can be used to read in virtually any kind of user input.

5.6.4 Text Box (textbox)

With this type of box, you can display the contents of text files. When called with this option, dialog starts a no-frills text file viewer that allows navigation with the arrow keys and searching for a string with /, as in the output of less and other programs.

Example:

```
dialog --textbox /etc/passwd 10 50
```

This displays the file `/etc/passwd` in a text box. The user can close the box by selecting the `EXIT` button:

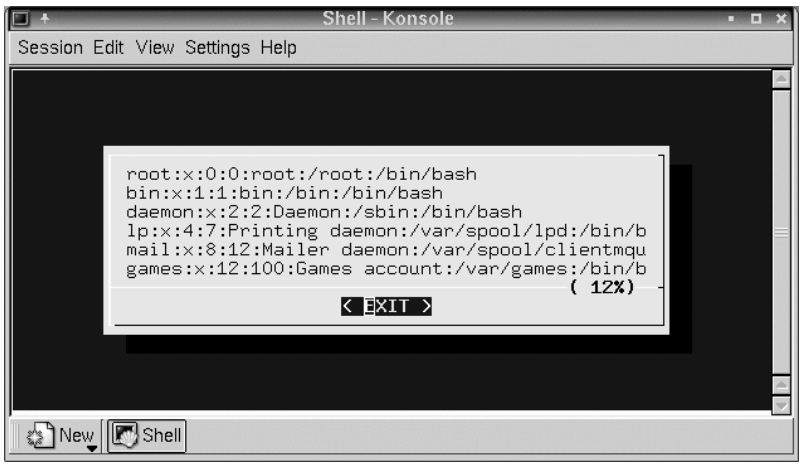


Figure 5.5: **Displaying `/etc/passwd` with a `textbox`**

To display the contents of a text file from a script, you can call `dialog` with the `--textbox` option.

5.6.5 Menu Box (`menu`)

With this type of box, the user can be presented with a menu to select one (and **only one**) item from a list. This example shows the syntax required for a `menu` box:

```
dialog --menu "This is a menu." 10 50 2 \
"option 1" "This is the first option" \
"option 2" "This is the second option"
```

The example shows that the `--menu` option also requires specification of the number of items (2 in the example) in addition to the height and width (10 and 50). As the first element of an item definition, `dialog` expects a name or tag for the item (an arbitrary string). This item name will be written to standard error if the corresponding item is selected. Standard error is redirected to a temporary file to allow querying the user's selection later.

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

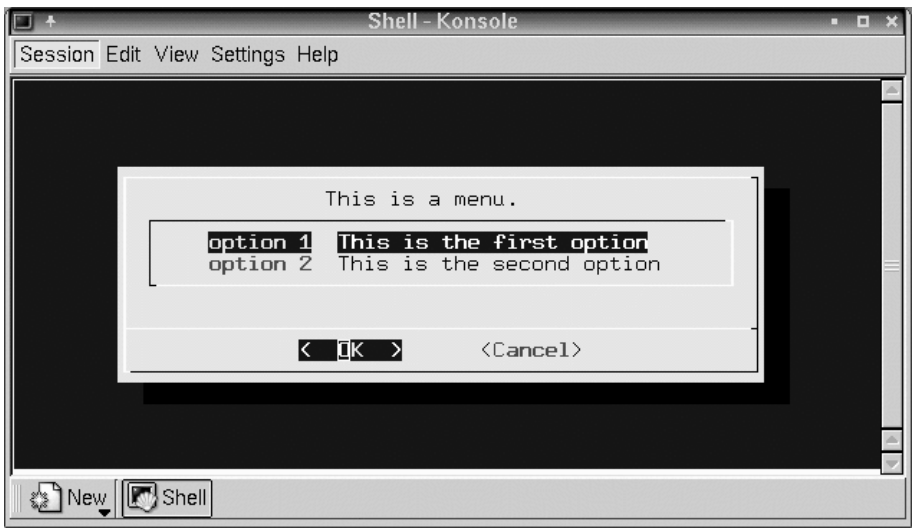


Figure 5.6: **A Menu Box Created with `dialog --menu`**

Given that the menu can also be closed with `Cancel`, it may be a good idea to check for the corresponding exit status to exit the script if necessary:

```
test "$?" = "1" && exit 0
```

A menu box is a good solution whenever there is a need to present the user with a list of items from which exactly one item should be selected.

5.6.6 Check List Box (`checklist`)

A check list box is similar to a menu box in that it presents a number of entries from which to select. However, any number of the check boxes in the list can be activated or deactivated (by pressing the space bar), which is different from a menu box and also from a radio list (see 5.6.7 on the next page) where only one item can be selected at a time.

The syntax is quite similar to that of the `--menu` option, but it allows one additional parameter, which is that each entry can have either "on" or "off" at the end of the corresponding line to indicate whether the entry should be set to on or off by default:

```
dialog --checklist "This is a checklist" 10 50 2 \
"a" "This is one option" "off" \
"b" "This is the second option" "on"
```

In this example, the second entry is on and the first one off by default, which results in the following dialog:

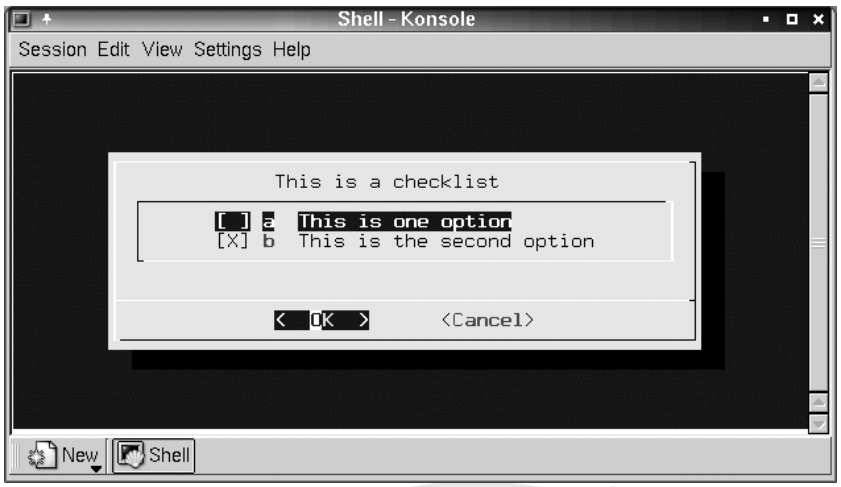


Figure 5.7: **A checklist Box**

Similarly to what happens with a menu box (see 5.6.5 on page 76), the names of any entries that are activated are put out on stderr and therefore need to be redirected to a temporary file. Assuming that both entries are activated in the above example and that redirection is actually performed in the way described, the contents of the temporary file will read `"a" "b"`. This can be processed for further use in a shell script with a `case` statement or through some other mechanism.

The box may be closed by selecting `Cancel`, so it is useful to intercept the corresponding status in the script (see Section 5.6.5 on page 76).

5.6.7 Radio List Box (`radiolist`)

The syntax to create a radio list with `dialog` is basically the same as the one for a check list, with the difference that only one of the entries can be active at a time and that entries are displayed as roundish button-like widgets (instead of square ones).

Example:

```
dialog --radiolist "This is a selective list, where only one \
option can be chosen" 10 50 2 \
"a" "This is the first option" "off" \
"b" "This is the second option" "on"
```

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

Radio buttons are not square but round, as can be seen in this screenshot:

Figure 5.8: A `radiolist` Dialog

Once again, each entry needs to be given a name, which is printed to stderr when selected. Like the two previous boxes, a radiolist box can be closed with `Cancel`.

5.6.8 Progress Meter Box(gauge)

This option allows you to display a gauge displaying the progress of a given process. To make this work, however, `dialog` requires that the output of the process is available in numerical format. The following script shows how this can be implemented:

```
#!/bin/bash
declare -i COUNTER=1
{
while   test $COUNTER -le 100
do
        echo $COUNTER
        COUNTER=COUNTER+1
        sleep 1
done
}| dialog --gauge "This is a progress bar" \
10 50 0
```

In this example, we first create a counter increased by 1 per loop iteration. Also, each iteration includes a pause of one second (`sleep 1`). Further, with each loop iteration `echo` sends the counter value to standard output. As `dialog` must be able to read this on standard input, we put the whole `while` loop in braces ({ ... }), thus making sure that it is not run in a subshell. This allows us to send the output through a pipe, which feeds it to `dialog --gauge` on stdin.

The command `dialog --gauge` reads numbers to display them as a progress bar indicating a percentage, but to do so it requires an additional parameter (apart from box height and width). This parameter specifies the initial percentage to be shown by the gauge ("0" in the example).

Figure 5.9: **A Progress Meter Created with `dialog --gauge`**

A gauge makes sense only if the output of the process to monitor is indeed available in the form of percentage values that can be displayed by the progress bar.

A sample script that implements all types of boxes that can be created with `dialog` is included in Appendix E on page 126.

Summary

- The `read` command can not only be used to assign single strings to a variable, but also to read in command output or file contents in a line-by-line fashion.
- If you anticipate that certain command sequences will be used more than once in a script or if you want to make a complex script easier to read and understand, consider defining shell functions for certain routines. A function normally comprises a part of a script and makes it available under a user-definable name, such that the script part may be executed simply by stating this name further below in the script.
- Use the bash built-in command `getopts` to easily extract command-line options for shell scripts. With the `getopts` command, you can tell the script which options it should recognize and which action should be triggered by a given option.

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

- Running processes can be aborted and stopped by sending them signals. To intercept signals sent to a script, use the `trap` built-in. The command allows you to define the signals that should be intercepted and the action that you want the script to take if the signal is sent.
- It is often desirable to present users with a number of alternatives when executing a script. This can be achieved with `select`, which presents alternatives in the form of a simple menu structure.
- With the `dialog` package installed as an add-on, scripts can be enhanced with a number of keyboard-aware graphical interface elements for the display and selection of text, items, etc., even if there is no access to an X server when running the script.

© 2004, SUSE LINUX AG (http://www.suse.com/training/)

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

Appendix

© 2004, SUSE LINUX AG (http://www.suse.com/training/)

suse

A Useful Commands for Shell Scripts

A.1 cat

When combined with the `here` operator (see Section 2.5.2 on page 18), the `cat` command is a good choice to output several lines of text from a script. In interactive use, the command is mostly run with a file name as an argument, in which case `cat` prints the file contents on standard output.

A.2 cut

The `cut` command can be used to cut out sections of lines from a file, so only the specified section is printed on standard output. The command is applied to each line of text as available in a file or on standard input. With `cut -f`, cut out text fields. `cut -c` works with the characters specified. Both single sections (characters or fields) and several sections can be specified. The default delimiter to separate fields from each other is a tab, but a different field separator can be specified with the `cut -d` option.

Examples:

```
tux@earth:~> cut -d : -f1 /etc/passwd
root
bin
daemon
lp
mail
games
```

This specifies that the field separator should be a colon. In every line of `/etc/passwd`, the field that comes before the first colon is taken and printed to stdout.

```
tux@earth:~> ls -l somedir/ | cut -c 35- | sort
  687 Sep 20 17:06 file2
 2199 Sep 20 17:05 file1
 6593 Sep 20 17:06 file3
```

This command takes the output of the `ls` command and cuts out everything from the thirty-fifth character. This is piped to `sort`, so the final output is sorted according to file size.

A.3 date

The `date` command can be used whenever there is a need to obtain a date or time string for further processing by a script. Without any options specified, the command's output looks like this:

```
tux@earth:~> date
Fre Sep 03 14:18:12 CEST 2002
tux@earth:~> su -
Password:
earth:~ # date
Fri Sep 03 14:18:34 CEST 2002
```

The example also shows that the output of `date` may differ from user to user ("Fre" vs. "Fri" for Friday), depending on how the locale variables have been set.

The `date` allows you to change the output format in almost every detail. With the `-I` option, `date` prints the date and time in ISO format (which is the same as if the options had been: `+%Y-%m-%d`.

```
tux@earth:~> date -I
2002-09-03
tux@earth:~> date +%m-%d' '%H:%M
09-03 14:19
tux@earth:~> date +%D,' '%r
09/03/02, 02:19:58 PM
tux@earth:~> date +%d.%m.%y
03.09.02
tux@earth:~> date +%d.%m.%Y
03.09.2002
tux@earth:~> date +%e.%-m.%y,' '%l.%M' '%p
3.9.02, 2.20 PM
tux@earth:~> date +%A,' '%e.' '%B' '%Y
Friday, 3. September 2002
```

A list with all the possible format options for `date` can be obtained with `man date`. In any case, you should be able to customize the output to exactly match the requirements of your script.

A.4 echo

The `echo` command, which exists both as a shell built-in and as an external command, prints text lines on standard output. A line break is inserted automatically after each line. When called with the `-e` option, `echo` accepts a number of additional options.

These are some of the special sequences recognized by `echo` when run with the `-e` option:

\a Output an alert (sounding the bell). This does not work in a `konsole` terminal, however.

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

\c Do not add a new line at the end of the output.

\n Add a new line (line break).

The `cat` command (see Appendix A.1 on page 85) should be preferred over `echo` to output a text file or several lines of text.

A.5 grep, egrep

The command `grep` and its variant `egrep` are used to search files for certain patterns, according to this syntax:

`grep` *`searchpattern filename ...`*

The command prints lines that contains the given search pattern. It is also possible to specify several files, in which case the output will not only print the matching line, but also the corresponding file names. Several options are available to specify that only the line number should be printed, for instance, or that the matching line should be printed together with leading and trailing context lines.

Search patterns can be supplied in the form of regular expressions (see Appendix B on page 95), although the bare `grep` command is limited in this regard. To search for more complex patterns, use the `egrep` command (or `grep -E`) instead, which accepts extended regular expressions. As a simple way to deal with the difference between the two variants, just make sure you use `egrep` in all of your shell scripts.

The regular expressions used with `egrep` need to be in accordance with the standard regex syntax. You can read about the details of this topic in the manual page of `grep`.

To avoid having special characters in search patterns interpreted by the shell, enclose the pattern in quotation marks (see Section 2.7 on page 23).

Example:

```
tux@earth:~> egrep (b|B)lurb file*
bash: syntax error near unexpected token ‘|’
tux@earth:~> grep "(b|B)lurb" file*
tux@earth:~> egrep "(b|B)lurb" file*
file1:blurb
filei2:Blurb
```

A.6 sed

The `sed` program is a stream editor — an editor used from the command line rather than interactively. An important detail to keep in mind is that `sed` performs text transformations

on a line-by-line basis. The commands available for `sed` can be specified either directly on the command line or in a special command script loaded by the program on execution.

The `sed` command requires the following syntax:
`sed 'editing-command' filename`

The available editing commands are single-character arguments. These are some of them:

`d` delete

`s` substitute (replace)

`p` output line

`a` append after

As with other commands, the output of `sed` normally goes to standard output, but can also be redirected to a file.

Each `sed` command must be preceded by an exact address or address range specifying the lines to which the editing command applies.

Apart from the single-character commands for text transformations, you can also specify options to influence the overall behavior of the `sed` program. These are some important command-line options for `sed`:

`-n`, `--quiet`, `--silent` By default, `sed` will print all lines on standard output after they have been processed. This option suppresses the output so `sed` only prints those lines for which the `p` editing command has been given to explicitly reenable printing.

`-e` *`command1`* `-e` *`command2`* `...` This option is necessary when specifying two or more editing commands. It must be inserted before each additional editing command.

`-f` *`filename`* With this option, a script file can be specified from which `sed` should read its editing commands.

For many editing commands, it is important to specify the exact line or lines that should be processed by the command. One of the more frequently used address labels is "`$`", which stands for the last line.

Examples:

```
tux@earth:~> sed -n '1,9p' somefile
```

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

This command has the effect that only lines 1 through 9 are printed on stdout.

```
tux@earth:~> sed '10,$d' somefile
```

This deletes everything from line 10 to the end of the file and also prints the first 9 lines of `somefile`.

You can use a regular expression to define the address or address range for an editing command. Regular expressions must be enclosed in forward slashes. If an address is defined with such an expression, `sed` processes every line that includes the given pattern.

Example:

```
tux@earth:~> sed -n '/Murphy.*/p' somefile
```

This prints all lines that have the pattern "Murphy.*" in them.

To negate an address, put an exclamation point before it. Also, if you want `sed` to perform several editing commands for the same address, you need to enclose the commands in braces: `sed '1,10{`*`command1 ; command2`*`}`.

As a general rule, editing commands need to be separated either by a line break or a semicolon. Finally, you can use a leading exclamation point to negate not only addresses, but also to negate editing commands. This can be used to tell `sed` that it should **not** perform the command on any lines matched by the address.

Example:

```
tux@earth:~> sed '/^#/!d' somefile
```

With this command, `sed` will not delete any lines that are commented (those with a leading "#" in them), but print them.

These are the most important editing commands of `sed`:

Command	Example	Editing action
`d`	`sed '10,$d' file`	Delete line.

`a`	`sed 'a\`
	`text\`
	`text' file`
		Append text after the specified line, with line breaks and backslashes included as shown in the example.

`i`	`sed 'i\`
	`text\`
	`text' file`
		Insert text before the specified line.

`c`	`sed '2000,$c\` `...'  file`	Replace specified lines with the text.

Command	Example	Editing action
s	sed s/*x*/*y*/*opt*.	Search and replace — the search pattern *x* is replaced with pattern *y*. The search and the replacement pattern are regular expressions in most cases and the search and replace behavior can be influenced through various options.
y	sed y/*abc*/*xyz*/	(*yank*) Replace every character from the set of source characters with the character that has the same position in the set of destination characters.

The following options can be used with s (search and replace):

I Do not distinguish between uppercase and lowercase.

g Replace globally wherever the search pattern is found in the line (instead of replacing only the first instance).

n Replace the *n*th matching pattern only.

p Print the line after replacing.

w *file* Write the resulting text to the specified file rather than printing it on stdout.

Examples:

```
tux@earth:~> sed 's/:/ /' /etc/passwd
```

This command replaces the first colon in each line with a space.

```
tux@earth:~> sed 's/:/ /g' /etc/passwd
```

This replaces all colons in all lines with a space.

```
tux@earth:~> sed 's/:/ /2' /etc/passwd
```

This replaces only the second colon in each line with a space.

```
tux@earth:~> sed -n 's/\([aeiou]\)/\1\1/Igp'
```

This replaces all single vowels with double vowels. The example shows how matched patterns can be referenced with "\1" if the search pattern is given in parentheses (which in turn have to be escaped). The "I" option ensures that sed ignores the case. "g" has the effect that characters are replaced globally. Finally, the "p" option tells sed to print all lines processed in this way.

```
tux@earth:~> sed 'y/ABCDEFGHIJKLMNOPQRSTUVWXYZ/abcdefghijklmnopqrstuvwxyz/'
```

This command changes all uppercase characters to lowercase.

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

A.7 test

The `test` command exists both as a built-in and as an external program. It is used to compare values and to check for files and their properties (whether a file exists, whether it is executable, and so on). If the tested condition is true, `test` returns an exit status of 0; if the condition is not true, the exit status is 1. In shell scripts, the command is used mainly to declare conditions to influence the operation of loops, branches, and other statements.

The syntax of `test` is:

`test` *`condition`*

Alternatively, a `test` command may be given in the form of `[␣`*`condition`*`␣]`. Note that a space (as indicated by the "␣") must always be inserted after the opening bracket and before the closing bracket.

The `test` command can be used for the following tasks:

- Testing whether a file exists. Some of the available options are:

Option	Description
`-e`	file exists
`-f`	file exists and is a regular file
`-d`	file exists and is a directory
`-x`	file exists and is executable

Example:

```
tux@earth:~> ls memo*
memo1  memo2
tux@earth:~> test -e memo1; echo $?
0
tux@earth:~> test -e memo3; echo $?
1
```

- Comparing two files. Some of the available operators are:

Operator	Description
`-nt`	newer than
`-ot`	older than
`-ef`	referring to the same inode (e.g., in the case of a hard link)

- Comparing two integers. The available operators are:

Operator	Description
`-eq`	equal
`-ne`	not equal
`-gt`	greater than
`-lt`	less than
`-ge`	greater than or equal
`-le`	less than or equal

- Testing strings.

Command	Description
`test -z` *`string`*	Exit status is 0 (true) if the string has zero length (is empty).
`test` *`string`*	Exit status is 0 (true) if the string has non-zero length (consists of at least 1 character).
`test` *`string1`* `=` *`string2`*	Exit status is 0 (true) if the strings are equal.
`test` *`string1`* `!=` *`string2`*	Exit status is 0 (true) if the strings are not equal.

- Combined tests.

Command	Description
`test !` *`condition`*	Exit status is 0 (true) if the condition is not true.
`test` *`cond1`* `-a` *`cond1`*	Exit status is 0 (true) if both conditions are true.
`test` *`cond1`* `-o` *`cond1`*	Exit status is 0 (true) if either condition is true.

More detailed information about `test` can be obtained with both `help test` and `man test` (the built-in `test` command and the external one have identical features).

A.8 `tr`

The `tr` command is used to translate (replace) or delete characters. It reads from standard input and prints the result on standard output. With `tr`, it is not only possible to replace regular characters or sequences of such characters, but also special characters like "`\t`"

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

(horizontal tab) or "\r" (return). A complete list of all special characters handled by tr is included in the manual page of the program.

Syntax:
tr *set1 set2*

The characters included in *set1* are replaced with the characters included in *set2*.

Example:

```
tux@earth:~> cat m-laws | tr a-z A-Z
MURPHY'S LAWS

IF ANYTHING CAN GO WRONG, IT WILL.
...
```

With a command like this one, all lowercase characters in a file are changed to uppercase and the result is printed.

If *set2* contains fewer characters than *set1*, the former is extended with the last character of the latter until both have the same length.

You can use tr to delete characters from the first set by entering tr -d *set1*. This will not translate anything, but only delete the ones included in *set1*, printing the rest to standard output.

Example:

```
tux@earth:~> VAR=‘echo $VAR | tr -d %‘
```

With this command, tr deletes the percent sign from the original value of VAR and the result is assigned as a new value to the same variable.

By entering a command like tr -s *set1 char*, you can also use tr to replace a set of characters with a single character.

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

B Regular Expressions

Regular expressions are strings consisting of metacharacters and literals (regular characters and numerals). In the context of regular expressions, metacharacters are those characters that do not represent themselves but have special meanings. They may act as placeholders for other characters or may be used to indicate a position in a string. Many commands (such as `egrep` and `sed`) rely on regular expressions for pattern matching. It is important to remember, however, that some metacharacters used by the shell for filename expansion have a meaning different from the one discussed here.

To learn more about the structure of regular expressions, read the corresponding manual page with `man 7 regex`. The following table presents the most important metacharacters and their meanings:

Character	Meaning	Example
`^`	beginning of the line	`^The`: `The` is matched if at the beginning of the line
`$`	end of the line	`eighty$`: `eighty` is matched if at the end of line
`\<`	beginning of the word	
`\>`	end of the word	`\<thing\>`: matches the whole word `thing`
`[abc]`	one character from the set	`[abc]`: matches any one of `a`, `b`, and `c`
`[a-z]`	any one from the specified range	`[a-z]`: matches any one character from `a` to `z`; `[-:+]`: any one of `-`, `:` and `+`
`[^xyz]`	none of the characters	`[^xyz]`: `x`, `y`, and `z` are not matched
`.`	any single character	`file.`: matches `file1` and `file2`, but not `file10`
`+`	one or more of the preceding expression	`[0-9]+`: matches any number
`*`	any number (including none) of preceding single character	`file.*` matches `file`, `file2`, and `file10`
{*n,m*}	the preceding expression `n` times at minimum and `m` times at maximum	`[0-9]{1,5}`: matches any one-digit to five-digit number

Character	Meaning	Example
|	the expression before or after	`file|File`: matches `file` and `File`
`()`	enclose alternatives for grouping with others	`(f|F)ile` matches `file` and `File`
`?`	zero or one of the preceding	`file1?2`: matches both `file2` and `file12`
`\`	escape the following metacharacter to remove its special meaning	`www\.suse\.de`: matches `www.suse.de`, literally (with the dot not being treated as a metacharacter); this is also necessary for parentheses, e.g., matching a parenthetical pattern would require the expression `\([a-zA-Z]+\)`

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

C Special Variable Substitution Operators for Bash

In Bash, you can use special variable substitution operators to assign different values to variables without having to rely on external commands. These special substitution operators allow changing variables by deleting certain patterns in their values and returning the rest, for instance. They also allow you to set a default for a variable to provide for situations where no value can be assigned to it.

The following variable substitutions are possible:

Substitution operator	Description
`${VAR-`*`value`*`}`	Returns *`value`* if the variable does not exist.
`${VAR=`*`value`*`}`	Assigns *`value`* to the variable and returns *`value`* if the variable does not exist.
`${VAR+`*`value`*`}`	Returns *`value`* if the variable exists.
`${#VAR}`	Returns the number of characters in the value of *VAR*.
`${VAR#`*`pattern`*`}`	Deletes the shortest part matched by *`pattern`* from the beginning of the variable's value and returns the rest.
`${VAR##`*`pattern`*`}`	Deletes the longest part matched by *`pattern`* from the beginning of the variable's value and returns the rest.
`${VAR%`*`pattern`*`}`	Deletes the shortest part matched by *`pattern`* from the end of the variable's value and returns the rest.
`${VAR%%`*`Muster`*`}`	Deletes the longest part matched by *`pattern`* from the end of the variable's value and returns the rest.

The substitution operators returning or setting a default value ("-", "=", and "+") can also be prefixed with a colon with the effect that substitution not only happens if the variable does not exist, but also if it exists but has a null value (is empty).

Examples:

```
tux@earth:~> echo $VAR

tux@earth:~> echo ${VAR-valyou}
valyou
tux@earth:~> echo $VAR
```

```
tux@earth:~> echo ${VAR=valyou}
valyou
tux@earth:~> echo $VAR
valyou
tux@earth:~> VAR=
tux@earth:~> echo ${VAR=valyou}

tux@earth:~> echo ${VAR:=valyou}
valyou
tux@earth:~> echo $VAR
valyou
tux@earth:~> echo ${VAR+ValYou}
ValYou
```


 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

D Debugging Shell Scripts

One way to debug a shell script is to use a built-in command like `set`, which allows you to set options to change the shell's behavior. The option relevant for debugging can be enabled with `set -x` either directly on the shebang line (`#!/bin/bash -x`) or by adding the `set -x` command to the script body. After enabling this option, the script will print each command in its exact interpreted form.

Example:

```
tux@earth:~> set -x
tux@earth:~> find ~ -name *.sh -exec mv {} bin \;
+ find /home/tux -name '*.sh' -exec mv '{}' bin ';'
```

As shown by the example, the command line is printed once again after having been entered in its full interpreted form. When not needed anymore, disable this behavior by entering `set +x`.

In many cases, seeing what the script's commands actually do will help you to discover the source of the error. If not, you may need to make sure that it is not hidden behind any variables by modifying the script in such a way that the values of variables get printed at run-time. The simplest way to achieve this is to insert the `echo` command before the corresponding variable (`echo $VARIABLE`). However, doing this for each and every variable just for debugging purposes (and removing the command afterwards) may be too much of an effort in some cases. A better solution would be to write a function that allows enabling or disabling the debugging output through a single `DEBUG` variable:

```
debug () {
        test "$DEBUG"= "yes" || return 1
        # check that there are no arguments
         test -z "$*" && return
         echo "Debug: $*"
}
```

This function can be called in a script together with each variable whose value should be printed. One could also add a hint (such as the name of the variable) to identify the debugging output, which will then be printed next to the value, e.g., `debug "MAX: $MAX"`. The function is ready for use after declaring a `DEBUG` variable at the beginning of the script. If `DEBUG` is set to "yes", the script will print the values of variables as it is run. If not, there is no such output.

With such a function, it becomes very easy to enable debugging for the given script and to quickly disable it when not needed anymore.

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

E Sample Scripts

Sample Script 1

hello.sh:

```
#!/bin/bash
# This script prints a "Hello world" greeting
# Author: Tux Penguin
# Created: 8/22/2002

echo -e "\aHello\nworld"

exit 0
```

Sample Script 2

name.sh:

```
#!/bin/bash
# This script reads the user’s first and last name
# and then prints a greeting with the full name.
# Author: Tux Penguin
# Created: 8/22/2002

echo "Please enter your first name:"

# first name gets assigned to variable FIRSTNAME
read FIRSTNAME

echo "Please enter your last name:"

# last name gets assigned to variable LASTNAME
read LASTNAME

# Now print the greeting:
echo "Welcome to the club, $FIRSTNAME $LASTNAME"

exit 0
```

Sample Script 3

name2.sh:

```
#!/bin/bash
# This scripts reads the user’s first and last name
# and then prints a greeting with this full name.
# Author: Tux Penguin
# Created: 8/22/2002

echo "Please enter your first name:"

# first name gets assigned to variable FIRSTNAME
read FIRSTNAME

echo "Please enter your last name:"

# last name gets assigned to variable LASTNAME
read LASTNAME

# create a new NAME variable
NAME="$FIRSTNAME $LASTNAME"

# Now print the greeting:
echo "Welcome back home, $NAME"

exit 0
```

Sample Script 4

find2.sh:

```
#!/bin/bash
# This script searches for files in the current directory.
# The user is prompted to enter a file name; if no name is
# entered, we search for the default value anyway, which is
# set to "*.bak"
# Author: Tux Penguin
# Created: 8/22/2002

echo "Please enter the file to be searched for (default is: *.bak):"

read FILE

find . -name "${FILE:="*.bak"}"

exit 0
```

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

Sample Script 5

sum.sh:

```
#!/bin/bash
# This script lets the user specify two whole numbers and
# then adds them together. All arithmetic formats possible
# under Bash are used, one after another.
# Author: Tux Penguin
# Created: 8/22/2002

# First we declare INTEGERS1,2 and SUM as integer variables,
# better do this here at the top as with most variables:
declare -i INTEGER1
declare -i INTEGER2
declare -i SUM

# read first integer
echo "Please enter first integer: "
read INTEGER1

# read second integer
echo "Please enter second integer: "
read INTEGER2

# this uses 'expr' for Bourne shell compatibility:
RESULT=‘expr $INTEGER1 + $INTEGER2‘

echo "The 'expr' command returns the result: $RESULT."

# this uses the Bash built-in 'let':
let RESULT="$INTEGER1 + $INTEGER2"

echo "The 'let' built-in returns the result: $RESULT."

# this uses a Bash-specific arithmetic expression:
RESULT=$[$INTEGER1 + $INTEGER2]

#or:
#RESULT=$(($INTEGER1 + $INTEGER2))

echo "Using an arithmetic expression in Bash, the result is: $RESULT."

# this one uses the variables declared as integers above:
SUM=INTEGER1+INTEGER2

echo "Using the variables declared as integers, the sum is: $SUM."

exit 0
```

© 2004, SUSE LINUX AG (http://www.suse.com/training/)

Sample Script 6

birthday.sh:

```
#!/bin/bash
# This script prompts the user to enter his date of
# birth. If today is the user’s birthday, we congratulate.
# Author: Tux Penguin
# Created: 8/22/2002

echo "Please enter your date of birth (YYYY-MM-DD, for instance 1973-12-21): "

# date of birth is assigned to the BIRTHDAY variable
read BIRTHDAY

# yank the year from the date of birth
BIRTHDAY=${BIRTHDAY#[0-9]*-}

# today’s date is assigned to variable TODAY
TODAY=‘date -I‘

# yank the year from today’s date as well
TODAY=${TODAY#[0-9]*-}

if     test "$BIRTHDAY" = "$TODAY"
then
       echo "Tada! Happy birthday to you! Nice presents awaiting you ..."
       STATUS=0
else
       echo "Sorry to disappoint you, no presents today ..."
       STATUS=1
fi

exit $EXITSTAT
```

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

Sample Script 7

This script can be implemented in two different ways. Either a check is performed to see whether the user has provided exactly one argument (`test $# -eq 1`), or whether the user has *not* provided exactly one argument (`test $# -ne 1`). Both possibilities are shown below.

file1a.sh:

```
#!/bin/bash
# This lets the user supply a file name as an argument,
# and we test whether the file exists or not.
# The script will abort with an error if the number of
# arguments is not equal to one.
# Author: Tux Penguin
# Created: 8/22/2002

if      test $# -ne 1
then
        echo "Please try again. Make sure you specify"
        echo "one file name only!"
        STATUS=1
else
        if      test -e $1
        then
                echo "The file $1 does exist."
                STATUS=0
        else
                echo "The file $1 doesn't exist."
                STATUS=2
        fi
fi

exit $STATUS
```

© 2004, SUSE LINUX AG (http://www.suse.com/training/)

file1b.sh:

```
#!/bin/bash
# This lets the user supply a file name as an argument,
# and we test whether the file exists or not.
# The script will abort with an error if the number of
# arguments is not equal to one.
# Author: Tux Penguin
# Created: 8/22/2002

if      test $# -eq 1
then
        if      test -e $1
        then
                echo "The file $1 does exist."
                STATUS=0
        else
                echo "The file $1 doesn't exist."
                STATUS=2
        fi
else
        echo "You must specify exactly one file name!"
        STATUS=1
fi

exit $STATUS
```

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

Sample Script 8

This enhances Sample Script 7 to send error messages to stderr. Also, the error message now contains a hint on how to use the script.

file2.sh:

```
#!/bin/bash
# This lets the user supply a file name as an argument,
# and we test whether the file exists or not.
# The script will abort with an error if the number of
# arguments is not equal to one.
# Author: Tux Penguin
# Created: 8/22/2002

# first store the name of this script (without the path!)
# in the SCRIPTNAME variable:
SCRIPTNAME=‘basename $0‘

if      test $# -ne 1
then
        echo "You must specify exactly one file name!" 1>&2
        echo "Usage: $SCRIPTNAME filename" 1>&2
        STATUS=1
else
        if      test -e $1
        then
                echo "The file $1 does exist."
                STATUS=0
        else
                echo "The file$1 doesn’t exist."
                STATUS=2
        fi
fi

exit $STATUS
```

Sample Script 9

This script prints a warning if the file system usage is 90 per cent or higher on one of the partitions. To achieve this, we need check the output of the `df` command for the range of numbers between "90" and "100". The right tool for this job is `egrep` (but not `grep`), because it allows us to specify the search pattern as a regular expression. Both `grep` and `egrep` return an exit status of zero if a match is found for the given pattern, and an exit status of one if there is no match.

df1.sh:

```
#!/bin/bash
# This script checks partitions to see whether 90%
# or more of the available space is used on any of them,
# and prints a warning if that is the case.
# Author: Tux Penguin
# Created: 8/22/2002

# quick and dirty method ;-)

if      df -P | egrep "9[0-9]%|100%"
then
        echo "At least one of the partitions uses 90%"
        echo "or more of its available space!"
        exit 1
else
        echo "None of the partitions use 90%"
        echo "or more of their available space."
        exit 0
fi
```

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

Sample Script 10

animals.sh:

```
#!/bin/bash
# This scripts asks the user to provide the name of
# an animal, and then tells him how many legs that
# animal has.
# Author: Tux Penguin
# Created: 8/22/2002

# prompt for an animal name with cat + 'here' operator
cat << EOF
'Tis nothin' but a riddle-rattle script that input ever begs:
"Thou shalt me name a beast or bug or pet, I tell how many legs."
EOF

read CREATURE

case    "$CREATURE" in
        [dD]dog | [cC]cat | [mM]ouse )
                echo "A $CREATURE has 4 legs."
                ;;
        [bB]ird | [hH]uman | [mM]onkey )
                echo "A $CREATURE has 2 legs."
                ;;
        [sS]pider )
                echo "A $CREATURE has 8 legs."
                ;;
        [fF]ly )
                echo "A $CREATURE has 6 legs."
                ;;
        [mM]ill[ei]pede | [cC]entipede )
                echo "I knew you would ask me 'bout this one."
                ;;
        * )
                echo "I haven't the faintest idea how many legs"
                echo "a(n) $CREATURE has."
                ;;
esac

exit 0
```

Sample Script 11

processing yes/no input:

```
case "$VARIABLE" in
        [yY] | [yY][eE][sS] | [yY] [eE] [aA] [hH] )
                ... ;;
        [nN] | [nN][oO] | [nN][oO][pP][eE] )
                ... ;;
        * )
                error message ;;
esac
```


 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

Sample Script 12

This script implements a `while` loop which iterates over the loop body 100 times. First we need to create a counter, which then gets counted up by 1 with each iteration. There are several ways to do arithmetic operations in a shell script, but in this case we declare the variables involved as integers, which allows us to do the math without any variable expansion or other substitution.

counter1.sh:

```
#!/bin/bash
# A script to iterate over a simple "while" loop 100 times.
# Author: Tux Penguin
# Created: 8/22/2002

# this declares the COUNTER variable as an integer
# which gets assigned the initial value of 1
declare -i COUNTER=1

while   test $COUNTER -le 100
do
        echo "The counter stands at $COUNTER."
        COUNTER=COUNTER+1
        sleep 1
done

exit 0
```

© 2004, SUSE LINUX AG (http://www.suse.com/training/)

Sample Script 13

This script implements an `until` loop to iterate over the loop body 100 times. First we need to create a counter, which then gets counted up by 1 with each iteration. There are several ways to do arithmetic operations in a shell script, but in this case we declare the variables involved as integers, which allows us to do the math without any variable expansion or other substitution.

counter2.sh:

```
#!/bin/bash
# A script to iterate over a simple \textt{until} loop 100 times.
# Author: Tux Penguin
# Created: 8/22/2002

# this declares the COUNTER variable as an integer
# which gets assigned the initial value of 1
declare -i COUNTER=1

until   test $COUNTER -gt 100
do
        echo "The counter stands at $COUNTER."
        COUNTER=COUNTER+1
        sleep 1
done

exit 0
```

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

Sample Script 14

The script's task is to replace uppercase with lowercase letters in all file names in the current directory. This is implemented with a `for` loop. The list of files is produced through command substitution, using the `find` command. The latter is executed with the `-maxdepth 1` option to prevent it from descending into subdirectories. We also need to make sure that the script only touches files with uppercase letters in their names.

However, in its current form the script is rather problematic because an existing file could be overwritten (for instance, if there were the files `memo` and `Memo`, renaming the latter to get an all-lowercase file name would also overwrite the original `memo` file).

lowercase1.sh:

```
#!/bin/bash
# This script renames all files in the current
# directory so that they have all-lowercase file names.
# Author: Tux Penguin
# Created: 8/22/2002

for     FILE in ‘find . -type f -maxdepth 1‘
do
        NEWFILE=‘echo $FILE | tr [A-Z] [a-z]‘
        if      test $FILE != $NEWFILE
        then
                echo mv $FILE $NEWFILE
        fi
done

exit 0
```

© 2004, SUSE LINUX AG (http://www.suse.com/training/)

Sample Script 15

This enhances Sample Script 14 to avoid overwriting any existing files.

lowercase2.sh:

```
#!/bin/bash
# This script renames all files in the current
# directory so that they have all-lowercase file names.
# 2nd version: Now we also check whether the file
# already exists with lowercase lettering.
# Author: Tux Penguin
# Created: 8/22/2002

for     FILE in ‘find . -type f -maxdepth 1‘
do
        NEWFILE=‘echo $FILE | tr [A-Z] [a-z]‘
        if      test $FILE != $NEWFILE
        then
                if      test -e $NEWFILE
                then
                        echo "There is already a file with the name $NEWFILE."
                        echo "$FILE will not be renamed."
                        # Skip the rest and begin next loop iteration:
                        continue
                fi
                echo mv $FILE $NEWFILE
        fi
done

exit 0
```

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

Sample Script 16

Sample Script 15 checked whether renaming a file would overwrite an existing file. If that is the case, the current loop iteration is terminated with a `continue` command. Thus the file in question is not renamed and keeps its uppercase letters.

It may therefore be useful to prompt the user for a different name. However, a user-supplied name could again lead to an existing file being overwritten. To avoid this, we use a `while` loop for the construct in question:

lowercase3.sh:

```
#!/bin/bash
# This script renames all files in the current
# directory so that they have all-lowercase file names.
# We also check whether the file already
# exists in lowercase lettering.
# 3rd version: If there is already a file in lowercase,
# we prompt the user for a different name.
# Author: Tux Penguin
# Created: 8/22/2002

for     FILE in ‘find . -type f -maxdepth 1‘
do
        NEWFILE=‘echo $FILE | tr [A-Z] [a-z]‘
        if      test $FILE != $NEWFILE
        then
                while   test -e $NEWFILE
                do
                        echo "There is already a file with the name $NEWFILE."
                        echo "Please enter a different name:"
                        read NEWFILE
                done
                echo mv $FILE $NEWFILE
        fi
done

exit 0
```

Sample Script 17

This script prints a warning if too much space is used on one of the system's partitions. The threshold to trigger a warning is 90 per cent, if the script is called without an argument. The warning message now also informs about the name of the partition and the actual percentage (this script builds on Sample Script 9, Appendix E on page 108).

df2.sh:

```
#!/bin/bash
# This script checks whether too much space is used
# on any partitions. The threshold percentage to trigger
# a warning can be supplied as an argument, but
# defaults to 90 if no argument is given.
# Author: Tux Penguin
# Created: 8/22/2002
if      test $# -eq 1
then
        THOLD=$1
else
        THOLD=90
fi

EXITSTAT=0

df -P | grep ^/dev | \
while   read PART SIZE USED FREE PERCENTAGE MPOINT
do
        # remove the per cent sign from the variable
        PERCENTAGE=${PERCENTAGE%\%}
        # or: PERCENTAGE=‘echo $PERCENTAGE | tr -d %‘

        if      test "$PERCENTAGE" -ge "$THOLD"
        then
                echo "File system usage on partition $PART (mount point"
                echo "$MPOINT) has reached as much as $PERCENTAGE per cent!"
                EXITSTAT=1
        fi
done

exit $EXITSTAT
```

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

Sample Script 18

For testing purposes, an "`echo`" is put before all important commands, such as `chown`, and `userdel`.

userdel.sh:

```
#!/bin/bash
# This script prompts for a user name and then deletes
# the corresponding account. Optionally, the user's
# home directory is deleted as well.
# Author: Tux Penguin
# Created: 8/22/2002

#yesno-Define function
yesno (){
        while   true
        do
                echo "$*"
                echo "Please answer by entering (y)es or (n)o:"
                read ANSWER
                case "$ANSWER" in
                        [yY] | [yY][eE][sS] )
                                return 0
                                ;;
                        [nN] | [nN][oO] )
                                return 1
                                ;;
                        * )
                                echo "I can't understand you over here."
                                ;;
                esac
        done
}

read -p "Delete which user? " USER

if     yesno "Also delete home directory of $USER?"
then
        HOME=yes
fi

if yesno "Really delete user $USER?"
then
        if test "$HOME" = yes
        then
                echo userdel -r $USER
        else
                HOME=$( grep $USER /etc/passwd | cut -d: -f6 )
                echo chown -R root.root $HOME
                echo userdel $USER
        fi
fi

exit 0
```

Sample Script 19

This enhances Sample Script 17 (E on page 116) by adding a help function.

df3.sh:

```
#!/bin/bash
# This script checks whether too much space is used
# on any partitions. The threshold percentage to
# trigger a warning can be supplied as an argument, but
# defaults to 90% if no argument is given.
# Author: Tux Penguin
# Created: 8/22/2002

SCRIPTNAME=‘basename $0‘
EXITSTAT=0

# declare showhelp function for help message
showhelp () {
cat <<-EOF
The $SCRIPTNAME script shows a warning about any
partitions on which too much space is being used,
according to a given limit. You can specify the limit
(threshold) to trigger such a warning as an argument.
For instance, to show a warning on all partitions
where more than 75% of the space is used, enter:
$SCRIPTNAME 75
The default threshold is 90%.
EOF
}

if      test $# -eq 1
then
        if      test "$1" -gt 0 -a "$1" -le 100
        then
                THOLD=$1
        else
                showhelp 1>&2
                exit 2
        fi
else
        THOLD=90
fi

df -P | grep ^/dev |
while   read PART SIZE USED FREE PERCENTAGE MPOINT
do
        # remove the per cent sign from the variable
        PERCENTAGE=${PERCENTAGE%\%}
        # or: PERCENTAGE=‘echo $PERCENTAGE | tr -d %‘

        if      test "$PERCENTAGE" -ge "$THOLD"
        then
                echo "File system usage on partition $PART (mount point"
                echo "$MPOINT) has reached as much as $PERCENTAGE per cent!"
                EXITSTAT=1
        fi
```

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

```
done

exit $EXITSTAT
```

The script uses the `here` operator to output several lines of text with the help of `cat`. This prints the text to the screen exactly as contained in the script, with one exception: As a special option of the operator, the "-" is put between the operator itself and the redirector (`cat «-EOF`), which suppresses the tabs used for indentation in the actual output.

Sample Script 20

This script enhances sample script 19 (Appendix E on page 118) by adding the ability to accept options from the command line.

df4.sh:

```
#!/bin/bash
# This script checks whether too much space is used
# on any partitions. Options include the warning
# threshold to be used, and whether to mail the
# output to root. If no threshold is specified, the
# script uses the default value of 90%.
# Author: Tux Penguin
# Created: 8/22/2002

SCRIPTNAME=‘basename $0‘
EXITSTAT=0

# this sets a variable to define a temporary file, with the
# shell’s PID as the filename extension
TEMPFILE=/tmp/df.$$

# declare showhelp function for help message
showhelp () {
cat <<-EOF
The $SCRIPTNAME script shows a warning about any
partition on which too much space is being used,
according to a given limit. It accepts these options:
        -h        show this help message
        -m        do not print warning, but mail it to root
        -t THOLD  threshold in per cent to trigger a warning
If no threshold is specified, the script will output
a warning for all partitions where 90% or more of the
space is used.
EOF
}

# extracts options & defines their actions
while getopts hmt: VAR
do
        case $VAR in
                h ) showhelp 1>&2
                    exit 0 ;;
                m ) DOMAIL=yes ;;
                t ) THOLD="$OPTARG" ;;
        esac
done
```

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

```
# THOLD mustn’t be empty
if      test -n "$THOLD"
then
        # threshold must be between 1 and 100,
        # if not it gets rejected
        if      test "$THOLD" -le 0 -o "$THOLD" -gt 100
        then
                showhelp 1>&2
                exit 2
        fi
else
        # if no threshold was specified, we assume a
        # default of 90
        THOLD=90
fi

df -P | grep ^/dev |
while   read PART SIZE USED FREE PERCENTAGE MPOINT
do
        # remove the per cent sign from the variable
        PERCENTAGE=${PERCENTAGE%\%}
        # or: PERCENTAGE=‘echo $PERCENTAGE | tr -d %‘

        if      test "$PERCENTAGE" -ge "$THOLD"
        then
                echo "File system usage on partition $PART (mount point"
                echo "$MPOINT) has reached as much as $PERCENTAGE%!" \
                >> $TEMPFILE
                EXITSTAT=1
        fi
done

# message only needs to be put out if there was
# indeed a partition above THOLD, so that a temp. file
# got created in the 1st place
if      test -e $TEMPFILE
then
        if      test "$DOMAIL" = "yes"
        then
                cat $TEMPFILE | mail -s "File syst. usage at least \
                $THOLD% on one partition!" root
        else
                cat $TEMPFILE
        fi
fi

# clean up
test -e $TEMPFILE && rm $TEMPFILE

exit $EXITSTAT
```

Sample Script 21

This modifies Sample Script 20 (Appendix E on page 120) to handle signals 2 and 15 in a clean way.

df5.sh:

```
#!/bin/bash
# This script checks whether too much space is used
# on any partitions. Options include the warning
# threshold to be used, and whether to mail the
# output to root. If no threshold is specified, the
# script uses the default value of 90%.
# Author: Tux Penguin
# Created: 8/22/2002

SCRIPTNAME=‘basename $0‘
EXITSTAT=0

# this sets a variable to define a temporary file, with the
# shell’s PID as the filename extension
TEMPFILE=/tmp/df.$$

# trap signals 2 (SIGINT, or ctrl+c), 15 (SIGTERM)
trap "rm $TEMPFILE; exit 123" 2 15

# declare showhelp function for help message
showhelp () {
cat <<-EOF
The $SCRIPTNAME script shows a warning about any
partition on which too much space is being used,
according to a given limit. It accepts these options:
        -h        show this help message
        -m        do not print warning, but mail it to root
        -t THOLD  threshold in per cent to trigger a warning
If no threshold is specified, the script will output
a warning for all partitions where 90% or more of the
space is used.
EOF
}

# extracts options & defines their actions
while getopts hmt: VAR
do
        case $VAR in
                h ) showhelp 1>&2
                    exit 0 ;;
                m ) DOMAIL=yes ;;
                t ) THOLD="$OPTARG" ;;
        esac
done
```

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

```
# THOLD mustn’t be empty
if      test -n "$THOLD"
then
        # threshold must be between 1 and 100,
        # if not it gets rejected
        if      test "$THOLD" -le 0 -o "$THOLD" -gt 100
        then
                showhelp 1>&2
                exit 2
        fi
else
        # if no threshold was specified, we assume a
        # default of 90
        THOLD=90
fi

df -P | grep ^/dev |
while   read PART SIZE USED FREE PERCENTAGE MPOINT
do
        # remove the per cent sign from the variable
        PERCENTAGE=${PERCENTAGE%\%}
        # or: PERCENTAGE=‘echo $PERCENTAGE | tr -d %‘

        if      test "$PERCENTAGE" -ge "$THOLD"
        then
                echo "File system usage on partition $PART (mount point \
                $MPOINT) has reached as much as $PERCENTAGE%!" \
                >> $TEMPFILE
                EXITSTAT=1
        fi
done

# message only needs to be put out if there was
# indeed a partition above THOLD, so that a temp. file
# got created in the 1st place
if      test -e $TEMPFILE
then
        if      test "$DOMAIL" = "yes"
        then
                cat $TEMPFILE | mail -s "File syst. usage at least \
                $THOLD% on one partition!" root
        else
                cat $TEMPFILE
        fi
fi

# clean up
test -e $TEMPFILE && rm $TEMPFILE

exit $EXITSTAT
```

Sample Script 22

groupmembers.sh:

```
#!/bin/bash
# This script takes a group name as an argument, and then
# filters the contents of /etc/passwd and /etc/group
# to list either the primary members, the secondary members,
# or all members of the given group.
# Author: Tux Penguin
# Created: 8/22/2002

SCRIPTNAME=‘basename $0‘

# abort if script has not been called with a
# group name as an argument
if [ $# -ne 1 ]
then
        echo "Usage: $SCRIPTNAME groupname"
        exit 1
fi

# the GROUP variable stores the name of the group
# in question, and GROUPID the corresponding ID
GROUP=$1
GROUPID=‘grep ^$GROUP /etc/group | cut -d: -f3‘


# 1st function: get primary group members only
# the default field separator for read is space or tab,
# so we need to re-assign IFS to use a colon instead
prim_groupies () {
IFS=:
cat /etc/passwd |       while   read LOGIN PASS USERID GROUP REST
                        do      if      test "$GROUPID" = "$GROUP"
                                then
                                        echo $LOGIN
                                fi
                        done
}

# 2nd function, get secondary group members only
secnd_groupies () {
grep ^$GROUP /etc/group | cut -d: -f4
}
# display a menu
PS3="Please enter your selection: "
select i in "Primary group members" \
            "Secondary group members" \
            "All group members" Quit
do
```

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

```
        if      test "$i" = "Quit"
        then
                break
        else    case $i in
                        "Primary group members" )
                        echo "$i of group $GROUP: "
                        prim_groupies
                        ;;

                        "Secondary group members" )
                        echo "$i of group $GROUP: "
                        secnd_groupies
                        ;;

                        "All group members" )
                        echo "$i of group $GROUP: "
                        prim_groupies
                        secnd_groupies
                        ;;
                esac
        fi
done

exit 0
```

Sample Script 23

dialog-sample.sh:

```
#!/bin/bash
# This script implements a number of different dialog boxes.
# Author: Tux Penguin
# Created: 8/22/2002

# dialog directs user input to stderr, therefore we need to
# redirect to a temporary file in order to do something with
# it later on
TEMPFILE=/tmp/dialog.$$

# trap the TERM signal and keyboard interrupts so that
# temp. files are removed
trap ’rm $TEMPFILE ; exit 2’ 2 15


# a yesno box
dialog --title "Demo: yesno" --yesno \
"A yesno box can query the user’s answer to a question.
The user can select either <Yes> or <No>. If <Yes> is
selected, the exit status of dialog is "0"; if <No> is
selected, the exit status is "1".\n
Do you want to continue?" 10 70

# the exit status is "1" if the user selects <No>,
# in which case we exit from the script
test "$?" = "1" && exit 0


# a msgbox
dialog --title "Demo: msgbox" --msgbox \
"A message box can be used to display some kind of information.
The user is supposed to confirm by selecting <OK>." 8 70


# an inputbox
dialog --title "Demo: inputbox" --inputbox \
"With an inputbox, it is possible to read in user input for
processing by a script. The string entered goes to stderr,
from where it can be directed to a temporary file, for
instance. If the user selects <Cancel>, dialog returns an
exit status of "1".

Please enter some words:" 15 70 2> $TEMPFILE

# the exit status is "1" if the user selects <Cancel>,
# in which case we exit from the script
test "$?" = "1" && exit 0

# open a msgbox to show the user what has just been entered
dialog --title "Demo: your input" --msgbox \
"Believe it or not, this is what you’ve just typed: \n
‘cat $TEMPFILE‘" 10 70
```

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

```
# a text box
dialog --title "Demo: textbox" --textbox ~/bin/dialog-textbox.txt 15 70

# a menu
dialog --title "Demo: menu" --menu \
"With the \"menu\" option of dialog, you can create simple menus
from which exactly one item may be selected. The \"name\" of the
item (such as \"Item 1\") is printed to stderr and needs to be
redirected somehow to be of any use." 15 70 2 \
"Item 1" "First selectable menu thingy" \
"Item 2" "Second selectable menu thingy" 2> $TEMPFILE

# the exit status is "1" if the user selects <Cancel>,
# in which case we exit from the script
test "$?" = "1" && exit 0

# open a msgbox to show the user which item has been selected
dialog --title "Demo: your menu selection" --msgbox \
"This is the menu item that you've just selected: \n
'cat $TEMPFILE' " 10 70


# a checklist
dialog --title "Demo: checklist" --checklist \
"The \"checklist\" option of dialog lets you display a list
of entries, each of which can be activated by the user with
<Space>. You can also set which entry should be on or off
by default. The \"names\" of any selected entries are written
to stderr." 15 70 3 \
"a" "I speak German." "on" \
"b" "I also speak English." "on" \
"c" "I speak Spanish, too." "off" 2> $TEMPFILE

# the exit status is "1" if the user selects <Cancel>,
# in which case we exit from the script
test "$?" = "1" && exit 0

# now open a msgbox to show the user what has been selected
dialog --title "Demo: your checklist selection" --msgbox \
"Your computer says that you've just selected these
entries: \n
'cat $TEMPFILE' " 10 70
```

```
# a radiolist
dialog --title "Demo: radiolist" --radiolist \
"With the \"radiolist\" option, dialog displays a list
from which exactly one entry may be selected. With an
additional option, you can set whether an entry should
be on or off by default. The \"name\" of the selected
entry is written to stderrr." 15 70 4 \
"First" "I want to have my cake." "off" \
"Second" "I want to eat my cake." "off" \
"Third" "I want to have your cake and eat it." "off" 2> $TEMPFILE

# the exit status is "1" if the user selects <Cancel>,
# in which case we exit from the script
test "$?" = "1" && exit 0

# now show what has been selected in a msgbox
dialog --title "Demo: your radiolist selection" --msgbox \
"With regard to cakes, your choice has been the following: \n
‘cat $TEMPFILE‘ " 10 70


# using gauge to create a progress bar

{
declare -i COUNTER=1
while test $COUNTER -le 100
do
        echo $COUNTER
        COUNTER=$COUNTER+1
        sleep 1
done
} | dialog --title "Demo: unstoppable progress with gauge" --gauge \
"The gauge option makes it possible to display a progress bar.
The script uses a loop to create numeric values, which can be
displayed by dialog in the form of a progress meter indicating
a percentage." 10 70 0
```

To make sure that the sample script can actually display a text with `dialog -textbox`, you should create the following file and store it as `~/bin/dialog-textbox.txt`:

```
You are looking at the contents of a plain text file, which
is displayed with the help of the textbox option of dialog.
A text box is dialog’s equivalent of a simple pager, with similar
features. For instance, the "/" key allows you to search for a
string in the text.
```

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)

Index

© 2004, SUSE LINUX AG (http://www.suse.com/training/)

 © 2004, SUSE LINUX AG (http://www.suse.com/training/)