Chapter
33

# TUTORIAL: SHELL SCRIPTING

Until now, you have been giving commands to the UNIX shell by typing them on the keyboard. When used this way, the shell is said to be a command interpreter. The shell can also be used as a high-level programming language. Instead of entering commands one at a time in response to the shell prompt, you can put a number of commands in a file, to be executed all at once by the shell. A program consisting of shell commands is called a *shell script*.

## 33.1 A Simple Shell Script

Suppose you were to make up a file named `commands` as shown in Listing 33-1.

```
① # A simple shell script
② cal
  date
  who
```

**Listing 33-1**
The `commands` script.

Let's dissect the `commands` script:

```
① # A simple shell script
```

The first line in this file begins with a # symbol, which indicates a *comment line*. Anything following the #, up to the end of the line, is ignored by the shell.

```
② cal
  date
  who
```

The remaining three lines are shell commands: the first produces a calendar for the current month, the second gives the current date and time, and the third lists the users currently logged onto your system.

We can get the Bourne Shell (`sh`) to run these commands by typing

```
§ sh < commands Return
```

The redirection operator (<) tells the shell to read from the file `commands` instead of from the standard input. It turns out, however, that the redirection symbol is not really needed in this case. Thus, you can also run the `commands` file by typing

```
§ sh commands Return
```

Is there any way to set up `commands` so that you can run it without explicitly invoking the shell? In other words, can you run `commands` without first typing `sh`? The answer is yes, but you first have to make the file *executable*. The `chmod` utility does this:

`chmod` is described in Chapter 6.

```
§ chmod u+x commands Return
```

The argument `u+x` tells `chmod` that you want to add (+) permission for the user (`u`) to execute (`x`) the shell script in the file. Now all you need do is enter the file name

```
§ commands Return
```

The search path is the list of directories where the shell is to look for executable files.

If your search path is set up to include the current directory, the shell will run the commands in the file. If it is not, the shell will complain that it cannot find the file you want it to execute:

```
commands not found
```

If this happens, you can still get the shell to run your commands this way:

```
§ ./commands Return
```

Remember, "dot" (`.`) stands for the current working directory.

## 33.2 Subshells

The new shell process is a subshell or child of the original shell.

When you tell the shell to run a script such as the `commands` file, your login shell actually calls up another shell process to run the script. (Remember, the shell is just a program, and UNIX can run more than one program at a time.) The parent shell waits for its child to finish, then takes over and gives you a prompt:

```
§
```

Incidentally, a subshell can be different from its parent shell. For example, you can have `ksh`, `csh`, `tcsh` or `bash` as your login shell, but use `sh` to run your shell scripts. Many users in fact do this. When it comes time to run a script, the login shell simply calls up `sh` as a subshell to do the job.

We will always use `sh` for running shell scripts. To make sure that `sh` is used, regardless of your login shell, include the following line at the top of each shell script file:

```
#!/bin/sh
```



The origins of the term "shebang" in this connection are obscure.

A pound sign and an exclamation point entered together (`#!`) as the first characters in the file are called the "shebang." Thus, we can modify our `commands` file as shown in Listing 33-2.

**Listing 33-2** The modified `commands` script.

```
#!/bin/sh
# A simple shell script
cal
date
who
```

This modified script can be run by entering its name at the shell prompt:

```
§ ./commands Return
```

## 33.3 The Shell as a Scripting Language

The sample script `commands` is almost trivial—it does nothing more than execute three simple commands that you could just as easily type into the standard input. The shell can actually do much more. It is, in fact, a sophisticated scripting language, with many of the same features found in other scripting languages, including

- Variables
- Input/output functions
- Arithmetic operations
- Conditional expressions
- Selection structures
- Repetition structures

We will discuss each of these in this chapter.

## 33.4 Variables

Three kinds of variables are commonly used in shell scripts:

- **Environment Variables.** Sometimes called special shell variables, keyword variables, predefined shell variables, or standard shell variables, they are used to tailor the operating environment to suit your needs. Examples include `TERM`, `HOME`, and `MAIL`.
- **User-created Variables.** These are variables that you create yourself.
- **Positional Parameters.** These are used by the shell to store the values of command-line arguments.

Of these, the environment variables have been introduced already, and the user-defined variables will be discussed later in this chapter. The positional parameters, which are very useful in shell programming, will be examined in this section.

The positional parameters are also called *read-only variables*, or *automatic variables*, because the shell sets them for you automatically. They "capture" the values of the command-line arguments that are to be used by a shell script. The positional parameters are numbered 0, 1, 2, 3, ... , 9. To illustrate their use, consider the following shell script, and assume that it is contained in an executable file named `echo.args`:

```
#!/bin/sh
# Illustrate the use of positional parameters
echo $0 $1 $2 $3 $4 $5 $6 $7 $8 $9
```

Suppose you run the script by typing the command line

```
§ echo.args We like UNIX. (Return)
```

The shell stores the name of the command ("`echo.args`") in the parameter `$0`; it puts the argument "`We`" in the parameter `$1`; it puts "`like`" in the parameter `$2`, and "`UNIX.`" in parameter `$3`. Since that takes care of all the arguments, the rest of the parameters are left empty. Then the script prints the contents of the variables:

```
echo.args We like UNIX.
```

What if the user types in more than nine arguments? The positional parameter `$*` contains all of the arguments `$1`, `$2`, `$3`, ... `$9`, and any arguments beyond these nine. Thus, we can rewrite `echo.args` to handle any number of arguments:

```
#!/bin/sh
# Illustrate the use of positional parameters
echo $*
```

The shell also counts the arguments that the user typed; this number is stored in the parameter `$#`. We can modify the script `echo.args` to use this parameter:

```
#!/bin/sh
# Illustrate the use of positional parameters
echo You typed $# arguments: $*
```

Suppose we were then to type the command line

```
§ echo.args To be or not to be (Return)
```

The computer would respond with

```
You typed 6 arguments: To be or not to be
§
```

`$*` does not contain `$0`, so `echo.args` is neither counted nor printed.



## 33.5 Making a File Executable: chex

If you are planning to write a lot of shell scripts, you will find it convenient to have a script that makes files executable. If we were to write a shell script for this, it might resemble Listing 33-3.

```
#!/bin/sh
# Make a file executable

① chmod u+x $1
② echo $1 is now executable:
  ls -l $1
```

**Listing 33-3**
The chex script.

Let's examine the interesting new features of the `chex` script:

```
① chmod u+x $1
```

Recall that the `chmod` utility changes file permissions. The utility takes two arguments. The first (`u+x`) adds execution privileges to the user. The second (`$1`) is a positional parameter that contains the name of the file.

```
② echo $1 is now executable:
  ls -l $1
```

The script confirms that the file permissions have been changed.

Next, we need to make the `chex` file executable. The easiest way to do this is to tell the shell to run `chex` on itself. Try this command line:

```
§ sh chex chex (Return)
```

This tells the shell to run `chex`, taking the `chex` file as the argument. The result is that `chex` makes itself executable. The output from this command will look something like this:

```
chex is now executable:
-rwxr-xr-x     1    yourlogin  59   Date time  chex
```

Now you can use `chex` to make other files executable.

## 33.6 The set Command

The positional parameters are sometimes called *read-only variables* because the shell sets their values for you when you type arguments to the script. However, you can also set their values using the `set` command. To see how this command works, consider the shell script shown in Listing 33-4, which we will assume is in the file `setdate`.

**Listing 33-4**
The setdate script.

```
#!/bin/sh
# Demonstrate the set command

① set `date`
② echo "Time: $4 $5"
  echo "Day: $1"
  echo "Date: $3 $2 $6"
```

This script introduces some new features:

```
① set `date`
```

The backquotes cause the date command to be run, with its output being captured by the set command and stored in the positional parameters $1 through $5.

```
② echo "Time: $4 $5"
  echo "Day: $1"
  echo "Date: $3 $2 $6"
```

Here is how we print out the values of the positional parameters.

Once setdate has been made executable by the chmod utility or the chex script, we can run the script by typing the command

```
§ ./setdate Return
```

The output will show the current time, day, and date:

```
Time: 10:56:08 EST
Day: Fri
Date: 20 Aug 2004
```

To understand what the script does, consider the command line

```
set `date`
```

The backquotes run the date utility, which produces output something like this:

```
Fri Aug 20 10:56:08 EST 2004
```

This does not appear on the screen. Instead, the set command catches the different parts of the output and stores them in the positional parameters $1 through $6:

```
$1 contains Fri
$2 contains Aug
$3 contains 20
$4 contains 10:56:08
$5 contains EST
$6 contains 2004
```



## 33.7 Labeling the Output from wc: mywc

The wc ("word count") filter counts the words, lines, and characters in a file. For example, try running wc on the chex file:

```
§ wc chex Return
5   17   84    chex
§
```

The output tells us that there are 5 lines, 17 words, and 84 characters in the file chex. This can be very useful information, but it would be a bit more convenient to use if the output were labeled. Listing 33-5 shows a shell script that does this.

**Listing 33-5**
The mywc script.

```
#!/bin/sh
# Label the output from wc

① set `wc $1`
② echo "File: $4"
  echo "Lines: $1"
  echo "Words: $2"
  echo "Characters: $3"
```

This script is similar to the previous one:

```
① set `wc $1`
```

The wc $1 command is run inside the backquotes. The output from this command is then captured and stored in positional parameters $1 through $4. Note that $1 initially receives the file name when the mywc script is run; then it receives the number of lines counted by the wc utility.

```
② echo "File: $4"
  echo "Lines: $1"
  echo "Words: $2"
  echo "Characters: $3"
```

As shown here, the values of the positional parameters are printed, properly labeled.

Run the chex script to make mywc executable. Then run mywc on an ASCII file. You might try the chex file:

```
§ ./mywc chex Return

File: chex
Lines: 5
Words: 17
Characters: 84
```

## 33.8 User-Defined Variables

You can create your own shell variable by writing its name. A variable name may include uppercase letters (A through Z), lowercase letters (a through z), numerals (0 through 9), and the underscore character (_). A variable name may not contain spaces or begin with a numeral.

There is no need to declare a variable's data type because the shell only works on character strings.

A string can be put into a variable using the assignment operator. For example, the assignment

```
first_var="This is a string"
```

This assignment stores the string `This is a string` in the variable `first_var`, overwriting any string that may already be in the variable. Note that no spaces are allowed around the assignment operator. Once the assignment is done, the value can be assigned to another variable:

```
second_var=$first_var
```

This assignment copies the string `This is a string` from `first_var` into `second_var`. As a result, both variables contain the same value. The dollar sign prefix $ indicates that the value of the variable is to be used. Suppose we were to omit the dollar sign:

```
second_var=first_var
```

This assignment copies the string `first_var` into `second_var`.

## 33.9 Input Using the read Statement

The positional parameters are useful for capturing command-line arguments but they have a limitation: once the script begins running, the positional parameters cannot be used for obtaining more input from the standard input. For this you have to use the `read` statement and a user-defined variable. Listing 33-6 shows how this is done.

```
#!/bin/sh
# Use positional parameters, user-defined variables, and
# the read command

① echo 'What is your name?'
② read name
③ echo "Well, $name, you typed $# arguments:"
④ echo $*
```

**Listing 33-6**
The `echo.args` script.

Let's examine the interesting features of this script:

① `echo 'What is your name?'`



In this script, the `echo` command prints a prompt on the standard input.

② `read name`

The `read` command obtains the user's response and stores it in the user-defined variable `name`.

③ `echo "Well, $name, you typed $# arguments:"`

To obtain the contents of the variable `name`, we use a dollar sign prefix ($). The positional parameter $# contains the count of command-line arguments that are entered when the script is executed.

④ `echo $*`

The positional parameter $* contains the command-line arguments that are entered when the script is executed

The script `echo.args` works something like this:

```
§ echo.args To be or not to be (Return)
```

The shell script would respond by prompting you for your name:

```
What is your name?
```

Suppose you were to type

```
Rumpelstiltskin (Return)
```

The computer would respond with

```
Well, Rumpelstiltskin, you typed 6 arguments:
To be or not to be
```

## 33.10 Arithmetic Operations Using the expr Utility

The shell is not intended for numerical work—if you need to do many calculations, you should consider a scripting language such as Perl or a programming language such as C, C++, Fortran, or Java. Nevertheless, the `expr` utility may be used to perform simple arithmetic operations on integers. (`expr` is not a shell command, but rather a separate UNIX utility; however, it is most often used in shell scripts.) To use it in a shell script, you simply surround the expression with backquotes. For example, let's write a simple script called `add` that adds two integers typed as arguments (Listing 33-7).

```
#!/bin/sh
# Add two numbers

① sum=`expr $1 + $2`
② echo $sum
```

**Listing 33-7**
The add script.

This script has two executable lines:

```
① sum=`expr $1 + $2`
```

Here we defined a variable `sum` to hold the result of the operation. (Note that spaces are required around the plus sign, but are not allowed around the equals sign.) The backquotes cause the `expr` utility to be run, adding the contents of the parameters `$1` and `$2`.

```
② echo $sum
```

The `echo` command is used to print the value of `$sum`. Note that the dollar sign prefix (`$`) is needed.

Make the `add` script executable, then type the following line:

```
§ add 4 3 (Return)
```

The first argument (4) is stored in `$1`, and the second (3) is stored in `$2`. The `expr` utility then adds these quantities and stores the result in `sum`. Finally, the contents of `sum` are echoed on the screen:

```
7
§
```

Next try the following line:

```
§ add 0.5 0.5 (Return)
```

The values 0.5 and 0.5 will not be recognized as numbers because they contain decimal points. You might see something like this:

```
expr: non-numeric argument
```

The `expr` utility only works on integers (i.e., whole numbers). It can perform addition (+), subtraction (-), multiplication (*), integer division (/), and integer remainder (%).

## 33.11 Control Structures

Normally, the shell processes the commands in a script sequentially, one after another in the order they are written in the file. Often, however, you will want to change the way that commands are processed. You may want to choose to run one command or another, depending on the circumstances; or you may want to run a command more than once.

To alter the normal sequential execution of commands, the shell offers a variety of control structures. There are two types of *selection structures*, which allow a choice between alternative commands:

- `if-then-elif ... else/fi`
- `case`



There are three types of *repetition* or *iteration structures* for carrying out commands more than once:

- `for`
- `while`
- `until`

## 33.12 The if Statement and test Command

The `if` statement lets you choose whether to run a particular command (or group of commands), depending on some condition. The simplest version of this structure has the general form

```
if condition
then
    command(s)
fi
```

When the shell encounters a structure such as this, it first checks to see whether the *condition* is true. If so, the shell runs any *command(s)* that it finds between the `then` and the `fi` (which is just *if* spelled backwards). If the *condition* is not true, the shell skips the *command(s)* between `then` and `fi`. A shell script that uses a simple `if` statement is shown in Listing 33-8.

**Listing 33-8**
The `friday` script.

```
#!/bin/sh

① set `date`

② if test $1 = Fri
then
   echo "Thank goodness it's Friday!"
fi
```

The `friday` script has some interesting features:

```
① set `date`
```

The `date` command is run (note the backquotes) and its output is captured by the `set` command.

```
② if test $1 = Fri
then
   echo "Thank goodness it's Friday!"
fi
```

Here we have used the `test` command in our conditional expression. The expression

```
test $1 = Fri
```

**Table 33-1**
Some arguments to the `test` command. Here, *file* represents the pathname of a file.

| Argument | Test is true if . . . |
|---|---|
| `-d` *file* | *file* is a directory |
| `-f` *file* | *file* is an ordinary file |
| `-r` *file* | *file* is readable |
| `-s` *file* | *file* size is greater than zero |
| `-w` *file* | *file* is writable |
| `-x` *file* | *file* is executable |
| `!` `-d` *file* | *file* is not a directory |
| `!` `-f` *file* | *file* is not an ordinary file |
| `!` `-r` *file* | *file* is not readable |
| `!` `-s` *file* | *file* size is not greater than zero |
| `!` `-w` *file* | *file* is not writable |
| `!` `-x` *file* | *file* is not executable |
| *n1* `-eq` *n2* | integer *n1* equals integer *n2* |
| *n1* `-ge` *n2* | integer *n1* is greater than or equal to integer *n2* |
| *n1* `-gt` *n2* | integer *n1* is greater than integer *n2* |
| *n1* `-le` *n2* | integer *n1* is less than or equal to integer *n2* |
| *n1* `-ne` *n2* | integer *n1* is not equal to integer *n2* |
| *n1* `-lt` *n2* | integer *n1* is less than integer *n2* |
| *s1* `=` *s2* | string *s1* equals string *s2* |
| *s1* `!=` *s2* | string *s1* is not equal to string *s2* |

checks to see if the parameter `$1` contains `Fri`; if it does, the `test` command reports that the condition is true, and the message is printed.

The `test` command can carry out a variety of tests; some of the arguments it takes are listed in Table 33-1.

## 33.13 The elif and else Statements

We can make the selection structures much more elaborate by combining the `if` with the `elif` ("else if") and `else` statements. The important thing to note about such structures is that no more than one of the alternatives may be chosen each time the selection structure is executed; as soon as one is, the remaining choices are skipped.

Listing 33-9 shows a script using an `if-then-elif...else-fi` structure.



**Listing 33-9**
The weekend script.

```
#!/bin/sh

set 'date'

① if test $1 = Fri
then
    echo "Thank goodness it's Friday!"
② elif test $1 = Sat  || test $1 = Sun
then
    echo "You should not be here working."
    echo "Log off and go home."
③ else
    echo "It is not yet the weekend."
    echo "Get to work!"
fi
```

The weekend script shows a three-part selection structure:

```
① if test $1 = Fri
then
    echo "Thank goodness it's Friday!"
```

Here, the first conditional expression is tested to see if the day is a Friday. If it is, the message "Thank goodness it's Friday!" is printed, and the shell script is finished.

```
② elif test $1 = Sat  || test $1 = Sun
then
    echo "You should not be here working."
    echo "Log off and go home."
```

If the first conditional is false, the second conditional expression is tested. Note that we have used the OR operator (`||`) in this expression to test whether the day is a Saturday or Sunday, in which case the second set of messages will be printed, and the script is finished.

```
③ else
    echo "It is not yet the weekend."
    echo "Get to work!"
fi
```

The `else` clause has no conditional; it is the *default case*, which is selected if no other pattern is matched. Thus, the third set of messages are printed if the other conditions are false. Note that the keyword `fi` terminates the selection structure.

We could make even more elaborate selection structures by including more `elif` clauses. Regardless of the number of alternatives in an `if-then-elif...else-fi` structure, no more than one will be selected each time the structure is executed. And once a choice is made, the remaining choices are skipped.

## 33.14 The case Statement

The shell provides another selection structure that may run faster than the `if` statement on some UNIX systems. This is the `case` statement, and it has the following general form:

```
case variable in
pattern1)   command(s) ;;
pattern2)   command(s) ;;
...
patternN)   command(s) ;;
esac
```

The `case` statement compares the value of *variable* with *pattern1*; if they match, the shell runs the *command(s)* controlled by that pattern. Otherwise, the shell checks the remaining patterns, one by one, until it finds one that matches the *variable*; it then runs the corresponding *command(s)*.

Listing 33-10 shows a simple shell script that uses the `case` statement instead of an `if-then-elif-else` structure.

```
#!/bin/sh

set `date`

① case $1 in
  Fri) echo "Thank goodness it's Friday!";;
② Sat | Sun) echo "You should not be here working";
             echo "Log off and go home!";;
③ *)    echo "It is not yet the weekend.";
        echo "Get to work!";;
  esac
```

**Listing 33-10**
The weekend2 script.

This script employs a three-part case structure:

```
① case $1 in
  Fri) echo "Thank goodness it's Friday!";;
```

If `$1` contains `Fri`, the message "Thank goodness it's Friday!" is printed, and the shell script is finished. The commands are separated by semicolons (`;`), and the end of a group of commands is indicated by two semicolons (`;;`).

```
② Sat | Sun) echo "You should not be here working";
             echo "Log off and go home!";;
```

We have used the OR operator (`|`) in this expression to test whether `$1` contains `Sat` or `Sun`, in which case the second set of messages will be printed, and the script is finished. Note that the OR symbol used in `case` statements is a single vertical line (|), not the double vertical lines (`||`) used in the `if` statement.



```
③ *)    echo "It is not yet the weekend.";
        echo "Get to work!";;
  esac
```

The pattern *) marks the *default case*, which is selected if no other pattern is matched. Thus, the third set of messages are printed if the other conditions are false.

## 33.15 for Loops

Sometimes we want to run a command (or group of commands) over and over. This is called *iteration*, *repetition*, or *looping*. The most commonly used shell repetition structure is the `for` loop, which has the general form

```
for variable in list
do
    command(s)
done
```

Here, *variable* is a user-defined variable—called the *control variable*—and *list* is a sequence of character strings separated by spaces. For each repetition of the loop, the control variable takes the value of the next item in the list and the *command(s)* in the body of the loop are executed. Here is a simple application of the `for` loop:

```
#!/bin/sh
#
for name in $*
do
   finger $name
done
```

Each time through the `for` loop, the control variable `name` takes on the value of the next argument in the list `$*`. This is then used as the argument to the `finger` command. Assuming this script is contained in the executable file `fingerall`, it would be run by typing the name of the file, followed by the login names you wish to finger:

```
§ fingerall johnp maryl frederick Return
```

## 33.16 while Loops

The general form of the `while` loop is

```
while condition
do
    command(s)
done
```

As long as the *condition* is true, the *command(s)* between the do and the done are executed. Here is an example of a shell script that uses the `expr` utility with the `while` loop to echo the keyboard entry ten times:

```
#!/bin/sh
# Print a message ten times
count=10
while test $count -gt 0
do
    echo $*
    count=`expr $count - 1`
done
```

## 33.17 until Loops

Another kind of iteration structure is the `until` loop. It has the general form

```
until condition
do
    command(s)
done
```

This loop continues to execute the *command(s)* between the do and done until the *condition* is true. We can rewrite the previous script using an `until` loop instead of the `while` loop:

```
#!/bin/sh
# Print a message ten times
count=10
until test $count -eq 0
do
    echo $*
    count=`expr $count - 1`
done
```

## 33.18 Removing Files Safely

The `rm` command can be very dangerous because it allows you to remove a file, but does not give you a way of getting back a file you may have removed accidentally. Most shells allow you to create an alias for the `rm` with the `-i` ("interactive") option; it will ask you if you are sure you want to remove the file in question. But `sh` does not allow aliases. Listing 33-11 shows a script that will duplicate the effect of the `rm -i` command. The script will also tell what action has been taken.



```
#!/bin/sh
# Delete a file interactively

① if test ! -f $1
then
    echo "There is no file \"$1\"."
② else
    echo "Do you want to delete \"$1\"?"
    read choice

③   if  test  $choice = y
    then
       rm $1
       echo "\"$1\" deleted."
④   else
       echo "\"$1\" not deleted."
⑤   fi
fi
```

**Listing 33-11**
The `del` script.

The `del` script has a selection structure nested within another selection structure:

```
① if test ! -f $1
  then
     echo "There is no file \"$1\"."
```

The script is designed take one command-line argument, the name of the file to be deleted. This file name is stored in the postitional parameter $1. The test

```
test ! -f $1
```

is true if the file named in $1 does *not* exist. In that case, the user is informed that there is no file by that name, and the script quits.

```
② else
     echo "Do you want to delete \"$1\"?"
     read choice
```

The default alternative is chosen when the file exists. In that case, the user is asked to confirm that he or she really wants to delete the file. The user's choice is read into the variable `choice`.

```
③    if  test  $choice = y
     then
        rm $1
        echo "\"$1\" deleted."
```

If the user's choice is `y`, the script calls the `rm` utility to remove the file, then prints an appropriate message.

```
④    else
        echo "\"$1\" not deleted."
```

If the user's choice is anything but `y`, the script takes no action other than printing an appropriate message.

```
⑤    fi
  fi
```

The first `fi` closes the inner selection structure; the second `fi` closes the outer structure.

## 33.19 An Improved Spelling Script

The `spell` utility is very useful, but it has a serious limitation: it lists the (possibly) misspelled words in a file, but does not tell you where in the file the misspelled words reside. Listing 33-12 describes a script that will correct this problem by labelling the output from the `spell` program.

```
#!/bin/sh
# An improved spelling-checker

① for word in `spell $1`
do
②    line=`grep -n $word $1`
     echo "     "
     echo "Misspelled word: $word"
③    echo "$line"
done
```

**Listing 33-12**
The `myspell` script.

If `spell` is not available, try `ispell` intead. Check the `man` pages for `ispell`.

The `myspell` script illustrates the use of a `for` loop:

```
① for word in `spell $1`
```

The `spell` utility is run on the file (note the backquotes), producing a list of (possibly) misspelled words. The loop variable `word` takes each of these misspelled words, one at a time.

```
②    line=`grep -n $word $1`
```

Here, `grep` is run on the file to find any lines containing the current misspelled word. The `-n` option causes `grep` to print the line number when a match is found.

```
③    echo "$line"
```

This command prints the contents of the variable `line`, which shows the current misspelled word in context.