27 – Flow Control: Branching with `if`

In the previous chapter, we were presented with a problem. How can we make our report-generator script adapt to the privileges of the user running the script? The solution to this problem will require us to find a way to "change directions" within our script, based on the results of a test. In programming terms, we need the program to *branch*.

Let's consider a simple example of logic expressed in *pseudocode*, a simulation of a computer language intended for human consumption.

X = 5

If X = 5, then:

    Say "X equals 5."

Otherwise:

    Say "X is not equal to 5."

This is an example of a branch. Based on the condition, "Does X = 5?" do one thing, "Say X equals 5," and otherwise do another thing, "Say X is not equal to 5."

`if`

Using the shell, we can code the previous logic as follows:

```
x=5

if [ "$x" -eq 5 ]; then
    echo "x equals 5."
else
    echo "x does not equal 5."
fi
```

Or we can enter it directly at the command line (slightly shortened).

```
[me@linuxbox ~]$ x=5
[me@linuxbox ~]$ if [ “$x” -eq 5 ]; then echo "equals 5"; else echo
"does not equal 5"; fi
equals 5
[me@linuxbox ~]$ x=0
[me@linuxbox ~]$ if [ “$x” -eq 5 ]; then echo "equals 5"; else echo
"does not equal 5"; fi
does not equal 5
```

In this example, we execute the command twice; once, with the value of `x` set to 5, which results in the string “equals 5” being output, and the second time with the value of `x` set to 0, which results in the string “does not equal 5” being output.

The `if` statement has the following syntax:

```
if commands; then
     commands
[elif commands; then
     commands...]
[else
     commands]
fi
```

where *commands* is a list of commands. This is a little confusing at first glance. But before we can clear this up, we have to look at how the shell evaluates the success or failure of a command.

Exit Status

Commands (including the scripts and shell functions we write) issue a value to the system when they terminate, called an *exit status*. This value, which is an integer in the range of 0 to 255, indicates the success or failure of the command’s execution. By convention, a value of zero indicates success and any other value indicates failure. The shell provides a parameter that we can use to examine the exit status. Here we see it in action:

```
[me@linuxbox ~]$ ls -d /usr/bin
/usr/bin
[me@linuxbox ~]$ echo $?
0
[me@linuxbox ~]$ ls -d /bin/usr
ls: cannot access /bin/usr: No such file or directory
[me@linuxbox ~]$ echo $?
2
```

In this example, we execute the `ls` command twice. The first time, the command executes successfully. If we display the value of the parameter `$?`, we see that it is zero. We execute the `ls` command a second time (specifying a nonexistent directory), producing an error, and examine the parameter `$?` again. This time it contains a 2, indicating that the command encountered an error. Some commands use different exit status values to provide diagnostics for errors, while many commands simply exit with a value of 1 when they fail. Man pages often include a section entitled "Exit Status," describing what codes are used. However, a zero always indicates success.

The shell provides two extremely simple builtin commands that do nothing except terminate with either a 0 or 1 exit status. The `true` command always executes successfully and the `false` command always executes unsuccessfully.

```
[me@linuxbox ~]$ true
[me@linuxbox ~]$ echo $?
0
[me@linuxbox ~]$ false
[me@linuxbox ~]$ echo $?
1
```

We can use these commands to see how the `if` statement works. What the `if` statement really does is evaluate the success or failure of commands.

```
[me@linuxbox ~]$ if true; then echo "It's true."; fi
It's true.
[me@linuxbox ~]$ if false; then echo "It's true."; fi
[me@linuxbox ~]$
```

The command `echo "It's true."` is executed when the command following `if` executes successfully and is not executed when the command following `if` does not execute successfully. If a list of commands follows `if`, the last command in the list is evaluated:

```
[me@linuxbox ~]$ if false; true; then echo "It's true."; fi
It's true.
[me@linuxbox ~]$ if true; false; then echo "It's true."; fi
[me@linuxbox ~]$
```

`test`

By far, the command used most frequently with `if` is `test`. The `test` command performs a variety of checks and comparisons. It has two equivalent forms. The first, shown here:

`test` *`expression`*

And the second, more popular form, shown here:

`[` *`expression`* `]`

where *expression* is an expression that is evaluated as either true or false. The `test` command returns an exit status of 0 when the expression is true and a status of 1 when the expression is false.

It is interesting to note that both `test` and `[` are actually commands. In `bash` they are builtins, but they also exist as programs in `/usr/bin` for use with other shells. The expression is actually just its arguments with the `[` command requiring that the `]` character be provided as its final argument.

The `test` and `[` commands support a wide range of useful expressions and tests.

File Expressions

Table 27-1 lists the expressions used to evaluate the status of files.

Table 27-1: `test` File Expressions

Expression	Is True If:
`file1` `-ef` *`file2`*	*file1* and *file2* have the same inode numbers (the two filenames refer to the same file by hard linking).
`file1` `-nt` *`file2`*	*file1* is newer than *file2*.
`file1` `-ot` *`file2`*	*file1* is older than *file2*.
`-b` *`file`*	*file* exists and is a block-special (device) file.
`-c` *`file`*	*file* exists and is a character-special (device) file.
`-d` *`file`*	*file* exists and is a directory.
`-e` *`file`*	*file* exists.
`-f` *`file`*	*file* exists and is a regular file.
`-g` *`file`*	*file* exists and is set-group-ID.
`-G` *`file`*	*file* exists and is owned by the effective group ID.

`-k` *`file`*	*file* exists and has its “sticky bit” set.
`-L` *`file`*	*file* exists and is a symbolic link.
`-O` *`file`*	*file* exists and is owned by the effective user ID.
`-p` *`file`*	*file* exists and is a named pipe.
`-r` *`file`*	*file* exists and is readable (has readable permission for the effective user).
`-s` *`file`*	*file* exists and has a length greater than zero.
`-S` *`file`*	*file* exists and is a network socket.
`-t` *`fd`*	*fd* is a file descriptor directed to/from the terminal. This can be used to determine whether standard input/output/error is being redirected.
`-u` *`file`*	*file* exists and is setuid.
`-w` *`file`*	*file* exists and is writable (has write permission for the effective user).
`-x` *`file`*	*file* exists and is executable (has execute/search permission for the effective user).

Here we have a script that demonstrates some of the file expressions:

```
#!/bin/bash

# test-file: Evaluate the status of a file

FILE=~/.bashrc

if [ -e "$FILE" ]; then
    if [ -f "$FILE" ]; then
         echo "$FILE is a regular file."
    fi
    if [ -d "$FILE" ]; then
         echo "$FILE is a directory."
    fi
    if [ -r "$FILE" ]; then
         echo "$FILE is readable."
    fi
    if [ -w "$FILE" ]; then
```

```
        echo "$FILE is writable."
    fi
    if [ -x "$FILE" ]; then
        echo "$FILE is executable/searchable."
    fi
else
    echo "$FILE does not exist"
    exit 1
fi

exit
```

The script evaluates the file assigned to the constant `FILE` and displays its results as the evaluation is performed. There are two interesting things to note about this script. First, notice how the parameter `$FILE` is quoted within the expressions. This is not required to syntactically complete the expression; rather it is a defense against the parameter being empty or containing only whitespace. If the parameter expansion of `$FILE` were to result in an empty value, it would cause an error (the operators would be interpreted as non-null strings rather than operators). Using the quotes around the parameter ensures that the operator is always followed by a string, even if the string is empty. Second, notice the presence of the `exit` command near the end of the script. The `exit` command accepts a single, optional argument, which becomes the script's exit status. When no argument is passed, the exit status defaults to the exit status of the last command executed. Using `exit` in this way allows the script to indicate failure if `$FILE` expands to the name of a nonexistent file. The `exit` command appearing on the last line of the script is there as a formality. When a script "runs off the end" (reaches end of file), it terminates with an exit status of the last command executed.

Similarly, shell functions can return an exit status by including an integer argument to the `return` command. If we were to convert the previous script to a shell function to include it in a larger program, we could replace the `exit` commands with `return` statements and get the desired behavior.

```
test_file () {

    # test-file: Evaluate the status of a file

    FILE=~/.bashrc

    if [ -e "$FILE" ]; then
        if [ -f "$FILE" ]; then
```

```
             echo "$FILE is a regular file."
        fi
        if [ -d "$FILE" ]; then
             echo "$FILE is a directory."
        fi
        if [ -r "$FILE" ]; then
             echo "$FILE is readable."
        fi
        if [ -w "$FILE" ]; then
             echo "$FILE is writable."
        fi
        if [ -x "$FILE" ]; then
             echo "$FILE is executable/searchable."
        fi
    else
        echo "$FILE does not exist"
        return 1
    fi

}
```

String Expressions

Table 27-2 lists the expressions used to evaluate strings:

Table 27-2: `test` String Expressions

Expression	Is True If...
`string`	*string* is not null.
`-n` *`string`*	The length of *string* is greater than zero.
`-z` *`string`*	The length of *string* is zero.
`string1 = string2` *`string1 == string2`*	*string1* and *string2* are equal. Single or double equal signs may be used. The use of double equal signs is supported by `bash` and is generally preferred, but it is not POSIX compliant.
`string1 != string2`	*string1* and *string2* are not equal.
`string1 > string2`	*string1* sorts after *string2*.
`string1 < string2`	*string1* sorts before *string2*.

Warning: the `>` and `<` expression operators must be quoted (or escaped with a backslash) when used with `test`. If they are not, they will be interpreted by the shell as redirection operators, with potentially destructive results. Also note that while the `bash` documentation states that the sorting order conforms to the collation order of the current locale, it does not. ASCII (POSIX) order is used in versions of `bash` up to and including 4.0. This problem was fixed in version 4.1.

Here is a script that incorporates string expressions:

```
#!/bin/bash

# test-string: evaluate the value of a string

ANSWER=maybe

if [ -z "$ANSWER" ]; then
    echo "There is no answer." >&2
    exit 1
fi

if [ "$ANSWER" = "yes" ]; then
    echo "The answer is YES."
elif [ "$ANSWER" = "no" ]; then
    echo "The answer is NO."
elif [ "$ANSWER" = "maybe" ]; then
    echo "The answer is MAYBE."
else
    echo "The answer is UNKNOWN."
fi
```

In this script, we evaluate the constant `ANSWER`. We first determine whether the string is empty. If it is, we terminate the script and set the exit status to 1. Notice the redirection that is applied to the echo command. This redirects the error message “There is no answer.” to standard error, which is the proper thing to do with error messages. If the string is not empty, we evaluate the value of the string to see whether it is equal to either “yes,” “no,” or “maybe.” We do this by using `elif`, which is short for “else if.” By using `elif`, we are able to construct a more complex logical test.

Integer Expressions

To compare values as integers rather than as strings, we can use the expressions listed in

Table 27-3.

Table 27-3: `test` Integer Expressions

Expression	Is True If...
integer1 `-eq` *integer2*	*integer1* is equal to *integer2*.
integer1 `-ne` *integer2*	*integer1* is not equal to *integer2*.
integer1 `-le` *integer2*	*integer1* is less than or equal to *integer2*.
integer1 `-lt` *integer2*	*integer1* is less than *integer2*.
integer1 `-ge` *integer2*	*integer1* is greater than or equal to *integer2*.
integer1 `-gt` *integer2*	*integer1* is greater than *integer2*.

Here is a script that demonstrates them:

```
#!/bin/bash

# test-integer: evaluate the value of an integer.

INT=-5

if [ -z "$INT" ]; then
    echo "INT is empty." >&2
    exit 1
fi

if [ "$INT" -eq 0 ]; then
    echo "INT is zero."
else
    if [ "$INT" -lt 0 ]; then
         echo "INT is negative."
    else
         echo "INT is positive."
    fi
    if [ $((INT % 2)) -eq 0 ]; then
         echo "INT is even."
    else
         echo "INT is odd."
    fi
fi
```

The interesting part of the script is how it determines whether an integer is even or odd. By performing a modulo 2 operation on the number, which divides the number by two and returns the remainder, it can tell whether the number is odd or even.

A More Modern Version of `test`

Modern versions of `bash` include a compound command that acts as an enhanced replacement for `test`. It uses the following syntax:

`[[ expression ]]`

where, like `test`, *expression* is an expression that evaluates to either a true or false result. The `[[ ]]` command is similar to `test` (it supports all of its expressions), but adds an important new string expression.

`string1 =~ regex`

This returns true if *string1* is matched by the extended regular expression *regex*. This opens up a lot of possibilities for performing such tasks as data validation. In our earlier example of the integer expressions, the script would fail if the constant `INT` contained anything except an integer. The script needs a way to verify that the constant contains an integer. Using `[[ ]]` with the `=~` string expression operator, we could improve the script this way:

```
#!/bin/bash

# test-integer2: evaluate the value of an integer.

INT=-5

if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
    if [ "$INT" -eq 0 ]; then
         echo "INT is zero."
    else
         if [ "$INT" -lt 0 ]; then
              echo "INT is negative."
         else
              echo "INT is positive."
         fi
         if [ $((INT % 2)) -eq 0 ]; then
              echo "INT is even."
         else
              echo "INT is odd."
         fi
```

```
    fi
else
    echo "INT is not an integer." >&2
    exit 1
fi
```

By applying the regular expression, we are able to limit the value of `INT` to only strings that begin with an optional minus sign, followed by one or more numerals. This expression also eliminates the possibility of empty values.

Another added feature of `[[ ]]` is that the `==` operator supports pattern matching the same way pathname expansion does. Here's an example:

```
[me@linuxbox ~]$ FILE=foo.bar
[me@linuxbox ~]$ if [[ $FILE == foo.* ]]; then
> echo "$FILE matches pattern 'foo.*'"
> fi
foo.bar matches pattern 'foo.*'
```

This makes `[[ ]]` useful for evaluating file and pathnames.

(( )) - Designed for Integers

In addition to the `[[ ]]` compound command, `bash` also provides the `(( ))` compound command, which is useful for operating on integers. It supports a full set of arithmetic evaluations, a subject we will cover fully in Chapter 34, "Strings and Numbers."

`(( ))` is used to perform *arithmetic truth tests*. An arithmetic truth test results in true if the result of the arithmetic evaluation is non-zero.

```
[me@linuxbox ~]$ if ((1)); then echo "It is true."; fi
It is true.
[me@linuxbox ~]$ if ((0)); then echo "It is true."; fi
[me@linuxbox ~]$
```

Using `(( ))`, we can slightly simplify the `test-integer2` script like this:

```
#!/bin/bash

```

```
# test-integer2a: evaluate the value of an integer.

INT=-5

if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
    if ((INT == 0)); then
         echo "INT is zero."
    else
         if ((INT < 0)); then
              echo "INT is negative."
         else
              echo "INT is positive."
         fi
         if (( ((INT % 2)) == 0)); then
              echo "INT is even."
         else
              echo "INT is odd."
         fi
    fi
else
    echo "INT is not an integer." >&2
    exit 1
fi
```

Notice that we use less-than and greater-than signs and that == is used to test for equivalence. This is a more natural-looking syntax for working with integers. Notice too, that because the compound command (( )) is part of the shell syntax rather than an ordinary command, and it deals only with integers, it is able to recognize variables by name and does not require expansion to be performed. We'll discuss (( )) and the related arithmetic expansion further in Chapter 34.

Combining Expressions

It's also possible to combine expressions to create more complex evaluations. Expressions are combined by using logical operators. We saw these in Chapter 17, "Searching for Files," when we learned about the find command. There are three logical operations for test and [[ ]]. They are AND, OR and NOT. test and [[ ]] use different operators to represent these operations :

Table 27-4: Logical Operators

Operation	test	[[ ]] and (( ))

AND	-a	&&
OR	-o	||
NOT	!	!

Here's an example of an AND operation. The following script determines whether an integer is within a range of values:

```
#!/bin/bash

# test-integer3: determine if an integer is within a
# specified range of values.

MIN_VAL=1
MAX_VAL=100

INT=50

if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
    if [[ "$INT" -ge "$MIN_VAL" && "$INT" -le "$MAX_VAL" ]]; then
         echo "$INT is within $MIN_VAL to $MAX_VAL."
    else
         echo "$INT is out of range."
    fi
else
        echo "INT is not an integer." >&2
        exit 1
fi
```

In this script, we determine whether the value of integer `INT` lies between the values of `MIN_VAL` and `MAX_VAL`. This is performed by a single use of `[[ ]]`, which includes two expressions separated by the `&&` operator. We could have also coded this using `test`:

```
    if [ "$INT" -ge "$MIN_VAL" -a "$INT" -le "$MAX_VAL" ]; then
         echo "$INT is within $MIN_VAL to $MAX_VAL."
    else
         echo "$INT is out of range."
    fi
```

The ! negation operator reverses the outcome of an expression. It returns true if an expression is false, and it returns false if an expression is true. In the following script, we modify the logic of our evaluation to find values of INT that are outside the specified range:

```
#!/bin/bash

# test-integer4: determine if an integer is outside a
# specified range of values.

MIN_VAL=1
MAX_VAL=100

INT=50

if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
    if [[ ! ("$INT" -ge "$MIN_VAL" && "$INT" -le "$MAX_VAL") ]]; then
         echo "$INT is outside $MIN_VAL to $MAX_VAL."
    else
         echo "$INT is in range."
    fi
else
        echo "INT is not an integer." >&2
        exit 1
fi
```

We also include parentheses around the expression, for grouping. If these were not included, the negation would only apply to the first expression and not the combination of the two. Coding this with test would be done this way:

```
    if [ ! \( "$INT" -ge "$MIN_VAL" -a "$INT" -le "$MAX_VAL" \) ];
then
         echo "$INT is outside $MIN_VAL to $MAX_VAL."
    else
         echo "$INT is in range."
    fi
```

Since all expressions and operators used by test are treated as command arguments by the shell (unlike [[ ]] and (( ))), characters that have special meaning to bash, such as <, >, (, and), must be quoted or escaped.

Seeing that `test` and `[[ ]]` do roughly the same thing, which is preferable? `test` is traditional (and part of the POSIX specification for standard shells, which are often used to run system startup scripts), whereas `[[ ]]` is specific to `bash` (and a few other modern shells). It's important to know how to use `test` since it is widely used, but `[[ ]]` is clearly more useful and is easier to code, so it is preferred for modern scripts.

Portability is the Hobgoblin of Little Minds

If you talk to "real" Unix people, you quickly discover that many of them don't like Linux very much. They regard it as impure and unclean. One tenet of Unix users is that everything should be "portable." This means that any script you write should be able to run, unchanged, on any Unix-like system.

Unix people have good reason to believe this. Having seen what proprietary extensions to commands and shells did to the Unix world before POSIX, they are naturally wary of the effect of Linux on their beloved OS.

But portability has a serious downside. It prevents progress. It requires that things are always done using "lowest common denominator" techniques. In the case of shell programming, it means making everything compatible with `sh`, the original Bourne shell.

This downside is the excuse that proprietary software vendors use to justify their proprietary extensions, only they call them "innovations." But they are really just lock-in devices for their customers.

The GNU tools, such as `bash`, have no such restrictions. They encourage portability by supporting standards and by being universally available. You can install `bash` and the other GNU tools on almost any kind of system, even Windows, without cost. So feel free to use all the features of `bash`. It's *really* portable.

Control Operators: Another Way to Branch

`bash` provides two control operators that can perform branching. The `&&` (AND) and `||` (OR) operators work like the logical operators in the `[[ ]]` compound command. Here is the syntax for `&&`:

```
command1 && command2
```

and here is the syntax for `||`:

```
command1 || command2
```

It is important to understand the behavior of these. With the `&&` operator, *command1* is

executed and *command2* is executed if, *and only if*, *command1* is successful. With the `||` operator, *command1* is executed and *command2* is executed if, *and only if*, *command1* is unsuccessful.

In practical terms, it means that we can do something like this:

```
[me@linuxbox ~]$ mkdir temp && cd temp
```

This will create a directory named `temp`, and if it succeeds, the current working directory will be changed to `temp`. The second command is attempted only if the `mkdir` command is successful. Likewise, a command like this:

```
[me@linuxbox ~]$ [[ -d temp ]] || mkdir temp
```

will test for the existence of the directory `temp`, and only if the test fails will the directory be created. This type of construct is handy for handling errors in scripts, a subject we will discuss more in later chapters. For example, we could do this in a script:

```
[ -d temp ] || exit 1
```

If the script requires the directory `temp` and it does not exist, then the script will terminate with an exit status of 1.

Summing Up

We started this chapter with a question. How could we make our `sys_info_page` script detect whether the user had permission to read all the home directories? With our knowledge of `if`, we can solve the problem by adding this code to the `report_home_space` function:

```
report_home_space () {
    if [[ "$(id -u)" -eq 0 ]]; then
         cat <<- _EOF_
              <h2>Home Space Utilization (All Users)</h2>
              <pre>$(du -sh /home/*)</pre>
              _EOF_
    else
         cat <<- _EOF_
```

```
            <h2>Home Space Utilization ($USER)</h2>
            <pre>$(du -sh $HOME)</pre>
            _EOF_
    fi
    return
}
```

We evaluate the output of the `id` command. With the `-u` option, `id` outputs the numeric user ID number of the effective user. The superuser is always ID zero and every other user is a number greater than zero. Knowing this, we can construct two different here documents, one taking advantage of superuser privileges, and the other restricted to the user's own home directory.

We are going to take a break from the `sys_info_page` program, but don't worry. It will be back. In the meantime, we'll cover some topics that we'll need when we resume our work.

Further Reading

There are several sections of the `bash` man page that provide further detail on the topics covered in this chapter:

- Lists (covers the control operators `||` and &&)
- Compound Commands (covers `[[ ]]`, `(( ))` and `if`)
- CONDITIONAL EXPRESSIONS
- SHELL BUILTIN COMMANDS (covers `test`)

Further, the Wikipedia has a good article on the concept of pseudocode:

http://en.wikipedia.org/wiki/Pseudocode