

3 - CONDITIONAL BRANCHING

- Unit Objectives:
 - Overview 3-2
 - Return values 3-3
 - Basic decision making: if blocks 3-4
 - The opposite of if: else blocks 3-5
 - Contracted nesting: elif 3-8
 - The test command 3-11
 - The test operators 3-14
 - Short circuit operators 3-17
 - The exit command 3-19
 - Command substitution: `command` 3-20
 - Simple Math: expr and $\$(())$ 3-21
 - Exercises 3-23
 - Exercise solutions 3-26

OVERVIEW

- In the previous chapters, we have examined the basic format of a shell script's content, that being any reasonable combination of:
 - Executable command lines
 - Comments
 - Whitespace
- In this chapter, we are going to take a first look at flow control within a shell script, learning how to execute certain commands only if and when a certain condition is TRUE (or maybe FALSE...).
- We will begin by examining the return values produced by several common commands and how these return values can be implemented as control factors, in which case these commands become "tested" commands also known as "governing" commands.
- We will then move on to look at a command that is used to explicitly test some value or values: the `test` command itself.

RETURN VALUES

- When a command is executed, it always return a value upon its completion.
- This return value is an integer in the range of 0 through 255.
- By convention, a value of 0 (zero) is indicative of success. Any non-zero return value is treated as failure.
- The return value is stored in the parent process's `?` variable. A sample sequence of commands shows how this works:

```
$ cat .kshrc
set -o vi
PS1='[$PWD]:$ '
export PS1
$ echo $?
0
$ cat nofile
cat: nofile: No such file or directory
$ echo $?
1
$ echo $?
0
```

- In the preceding example, the first execution of the `cat` command was successful. A subsequent examination of the value stored in the `?` variable shows a return value of 0. The second `cat` command was unsuccessful and returned a non-zero value (in this case, 1). Notice that executing the same `echo` command line again returned a 0. Why? Because the preceding command was successful (the previous `echo` command).

BASIC DECISION MAKING: if BLOCKS

- The Unix shell has a number of built in keywords and commands, some of which are used to form **constructs**.
- Perhaps the most common construct is the "if block".
- An "if block" uses these required keywords: `if`, `then` and `fi` (the reverse spelling of `if`).
- Here's a generic example of the syntax. The required keywords appear in **bold**:

```
if command1  
then  
    command2  
    command3  
    . . .  
fi
```

- If *command1* returns a 0 value, then *command2* and *command3* will execute, otherwise execution skips forward to continue with any commands that come after the `fi` keyword (the rest of the script).
- Several commands may appear between the keywords `if` and `then`, but it is the last one (immediately before `then`) that is the **tested** (or "**governing**") command.

THE OPPOSITE OF if: else BLOCKS

- There is an optional keyword, `else`, that allows commands to be executed if `command1` fails:

```
if command1
then
    command2
    command3
    ...
else
    command4
    command5
    ...
fi
```

- When a command is a **tested command** (as was the case with `command1` in the previous example) it will be viewed as having returned a TRUE value if it returns a 0.
- Here's an example script that uses `grep` (see the Appendix for an overview of `grep`) as the tested command. `grep` will return 0 if it finds the pattern it's looking for. `grep` returns a non-zero value if it doesn't find the pattern or if it can't open one of its input files. Here's the code for a script called `wednesday` that prints one of two messages based on whether it is or isn't Wednesday:

```
#!/usr/bin/ksh
if date | grep Wed > /dev/null
then
    echo "It's Wednesday!"
else
    echo "It's NOT Wednesday!"
fi
```

THE OPPOSITE OF `if`: `else` BLOCKS (continued)

- Here is the sequence of events that occur in the preceding shell code:
 1. The shell recognizes the built in keyword `if`, causing it to search forward through the remainder of the code looking for the other required keywords (`then` and `fi`).
 2. Having found the required keywords, the shell then executes both the `date` and `grep` commands simultaneously, with `date`'s output piped into `grep`'s input (the pipe is indicated by the `|` symbol).
 3. `grep` has been given the pattern `wed` to search for. By default, `grep` reads one or more input files and prints any records that contain a pattern, specified in the form of a regular expression as the first argument. In the absence of named input files, `grep` reads the standard input and that's what it's doing here. The trailing `> /dev/null` code simply discards any standard output produced by `grep` because we're only interested in whether it found the pattern or not.
 4. If `grep` is successful in finding the pattern `wed` in its input, `grep` returns a `0`, otherwise it returns a non-zero value.
 5. If a `0` is returned, the first `echo` command is executed, otherwise the second `echo` command is executed.

THE OPPOSITE OF if: else BLOCKS (continued)

- This same code could actually be entered on the command line, as the shell waits for the final closing keyword `fi` (the `>` is the shell's secondary prompt):

```
$ if date | grep Wed > /dev/null
> then
>     echo "It's Wednesday!"
> else
>     echo "It's NOT Wednesday!"
> fi
It's NOT Wednesday
$
```

CONTRACTED NESTING: `elif`

- There is another optional keyword, `elif`, that is a contraction of `else` and `if`. It allows for more readable code when you would otherwise have "nested `if`'s". Here's the generic syntax. Note that the `elif command` then block may be repeated any number of times:

```
if command1
then
    command2
    . . .
elif command3
then
    command4
    . . .
    .
    .
    .
else
    command5
    . . .
fi
```

CONTRACTED NESTING: elif (continued)

- Here's an example of nested `if`'s (note that the additional indenting of each nested `if` is not required; it only makes for more readable code). The file is called `NestedIfs`:

```
#!/usr/bin/ksh
if date | grep Mon > /dev/null
then
    echo "It's Monday"
else
    if date | grep Tue > /dev/null
    then
        echo "It's Tuesday"
    else
        if date | grep Wed > /dev/null
        then
            echo "It's Wednesday"
        else
            if date | grep Thu > /dev/null
            then
                echo "It's Thursday"
            else
                if date | grep Fri > /dev/null
                then
                    echo "It's Friday"
                fi
            fi
        fi
    fi
fi
```

CONTRACTED NESTING: elif (continued)

- Here's the same example using `elif`'s. This file is called `Elifs`:

```
#!/usr/bin/ksh
if date | grep Mon > /dev/null
then
    echo "It's Monday"
elif date | grep Tue > /dev/null
then
    echo "It's Tuesday"
elif date | grep Wed > /dev/null
then
    echo "It's Wednesday"
elif date | grep Thu > /dev/null
then
    echo "It's Thursday"
elif date | grep Fri > /dev/null
then
    echo "It's Friday"
fi
```

- By the way, in case you were wondering, there IS a better way to handle multiple choice scenarios like the one above. It's covered later in this document (it's another shell built in keyword construct called `case`).

THE test COMMAND

- In the preceding examples, `grep` was the **tested** (or **governing**) command. Often, however, we simply want to perform some type of value test. Unix provides the `test` command for just that reason.
- The history of `test` is rather lengthy. It has evolved over the years, and is summarized by this sequence that uses a string equivalence test syntax that is comparing the variable called `name` against the literal string `John Doe`:
 - Early Bourne shell (`sh`) uses the word `test` that can be used for string comparison, numeric comparison and file attribute tests. Test arguments and operators must be whitespace separated :

```
$ test "$name" = "John Doe"
```

- Later Bourne shell (`sh`) allows an alternate notation, substituting the square brackets (`[]`) for the word `test`. The brackets must be separated from the test arguments by whitespace. Also used for string, numeric and file attribute tests :

```
$ [ "$name" = "John Doe" ]
```

- Korn shell (`ksh`) and "Bourne Again" shell (`bash`) support both of the Bourne shell mechanisms as well as a more powerful double-square bracket notation (same whitespace requirements as the single-square brackets) that is used primarily for string and file attribute tests **only** :

```
$ [[ "$name" = "John Doe" ]]
```

THE test COMMAND (continued)

- Korn shell (`ksh`) and "Bourne Again" shell (`bash`) also have a double-parenthesis syntax that is used for numeric tests only. The comparison operators themselves are different than those used for string comparisons. The surrounding whitespace rules are waived. These two tests both perform a numeric equivalence test between the value of `$number` and the literal number `31`:

```
$ ( ( "$number" == 31 ) )
```

or

```
$ ( (" $number" == 31 ) )
```

- Regardless of which syntax you use, the `test` command does NOT produce any output (unless a critical error occurs). It merely returns a value: `0` for TRUE and `1` for FALSE, which makes it well suited for use in conjunction with the `if` constructs we examined earlier.

THE test COMMAND (continued)

- Whenever a shell variable is used as an argument to `test`, the shell variable substitution should be placed within double quotes. The reason: If the variable doesn't have any value, it will be replaced by a null string, which when unquoted, is not recognized as a command line argument. It just "disappears" into the surrounding whitespace. As such, `test` will usually generate a fatal error and issue a message indicating either a "missing argument" or "operator found where argument expected". A quoted null string at least appears as an empty argument, which satisfies the command line syntax requirements of `test`.

The moral of the story:

There is a BIG difference between **the presence of nothing** and the **absence of something**, as far as command line arguments are concerned!

THE test OPERATORS

- Here is a **partial** list of test operators and which constructs (STYLES) accept them (see the documentation for `test` and for your shell for the full list):

| <u>OPERATOR</u> | <u>DESCRIPTION</u> | <u>TYPE</u> | <u>STYLE</u> |
|-----------------|-------------------------------------|-------------|--------------|
| ➤ = | string equal | B | 1,2,3 |
| ➤ != | string not equal | B | 1,2,3 |
| ➤ -eq | numeric equal | B | 1,2,3 |
| ➤ -ne | numeric not equal | B | 1,2,3 |
| ➤ -lt | numeric less than | B | 1,2,3 |
| ➤ -gt | numeric greater than | B | 1,2,3 |
| ➤ -le | numeric less than or equal | B | 1,2,3 |
| ➤ -ge | numeric greater than or equal | B | 1,2,3 |
| ➤ -z | string length zero | U | 1,2,3 |
| ➤ -n | string length non-zero* | U | 1,2,3 |
| ➤ -e | file exists | U | 1,2,3 |
| ➤ -d | file exists and is a directory file | U | 1,2,3 |
| ➤ -f | file exists and is a regular file | U | 1,2,3 |
| ➤ -r | file exists and is readable | U | 1,2,3 |
| ➤ -w | file exists and is writable | U | 1,2,3 |
| ➤ -x | file exists and is executable | U | 1,2,3 |
| ➤ == | numeric equal | B | 4 |
| ➤ <> | numeric not equal | B | 4 |
| ➤ > | numeric greater than | B | 4 |
| ➤ < | numeric less than | B | 4 |
| ➤ >= | numeric greater than or equal | B | 4 |
| ➤ <= | numeric less than or equal | B | 4 |

*String length non-zero is the implied unary test if no operator is specified.

TYPES:

B Binary FORMAT: `arg1 operator arg2`

U Unary FORMAT: `operator arg1`

STYLES:

1. `test`
2. `[]`
3. `[[]]` (ksh and bash only)
4. `(())` (ksh and bash only)

THE test OPERATORS (continued)

- The numeric tests described in this chapter are **INTEGER TESTS ONLY!** None of the popular shells yet have support for performing floating point numeric tests.
- Here are some examples of the various forms of test syntax and the result assuming these given facts:
 - `$s1` has a string value of `alpha`.
 - `$s2` has a string value of `bravo`.
 - `$n1` has a numeric value of `10`.
 - `$n2` has a numeric value of `11`.
 - `$file` has a string value of `/etc/passwd` that is a readable, regular existing file.

| <u>TEST</u> | <u>RESULT</u> |
|---|---------------|
| ➤ <code>test "\$s1" = "\$s2"</code> | FALSE |
| ➤ <code>["\$s1" != "\$s2"]</code> | TRUE |
| ➤ <code>[[-z "\$s2"]]</code> | FALSE* |
| ➤ <code>test "\$n1" -gt "\$n2"</code> | FALSE |
| ➤ <code>["\$n1" -le "\$n2"]</code> | TRUE |
| ➤ <code>[["\$n1" -eq "\$n2"]]</code> | FALSE* |
| ➤ <code>test -d "\$file"</code> | FALSE |
| ➤ <code>[-f "\$file"]</code> | TRUE |
| ➤ <code>[[-e "\$file"]]</code> | TRUE* |
| ➤ <code>(("\$n1" < "\$n2"))</code> | TRUE* |
| ➤ <code>(("\$n1" == "\$n2"))</code> | FALSE* |

* ksh and bash only

THE test OPERATORS (continued)

- There are also some logical operators:

| <u>OPERATOR</u> | <u>DESCRIPTION</u> | <u>TYPE</u> | <u>STYLE</u> |
|-----------------|--------------------|-------------|--------------|
| ➤ -a | and | B | 1,2 |
| ➤ -o | or | B | 1,2 |
| ➤ ! | not | U | 1,2,3,4 |
| ➤ && | and | B | 3,4 |
| ➤ | or | B | 3,4 |

TYPES:

B Binary FORMAT: arg1 operator arg2

U Unary FORMAT: operator arg1

STYLES:

1. test
2. []
3. [[]] (ksh and bash only)
4. (()) (ksh and bash only)

- There is precedence associated with all of these logical operators. For example, "and" conditions are evaluated prior to "or" conditions.
- Parenthesis may be used either to override precedence or simply for readability, however they must be "escaped" using backslashes AND the parenthesis must be "whitespace separated" from the test arguments:

```
$ x=3
$ y=7
$ z=5
$ test $x -eq 3 -o $y -eq 8 -a $z -eq 9
$ echo $?
0
$ test \( $x -eq 3 -o $y -eq 8 \) -a $z -eq 9
$ echo $?
1
```

SHORT CIRCUIT OPERATORS

- The `&&` and `||` described on the previous page as "logical and" and "logical or" operators are also supported by the shell itself (as opposed to the `test` command, which was what the last few pages were talking about).
- When used alone on a command line (that is, when not part of an explicit `test` argument list) they act to separate individual command lines. The resultant "big" command line is sometimes called a "compound" command line, as it is made up of more than one individual command line.
- The reason they're called "short circuit" operators is that whenever the shell "sees" these operators, it knows that it must arrive at a logical conclusion. The `&&` presents a "logical and" scenario, and the `||` presents a "logical or". This forces the shell to examine the return values of the individual command lines that are separated by these logical operators in a Boolean (true or false) context.
- As soon as the shell has executed enough of the individual command lines to have arrived at a definite logical outcome, no more commands are executed, hence the term "short circuit" meaning "take the easy way out; do nothing that's unnecessary".

SHORT CIRCUIT OPERATORS (continued)

- Here are some examples:

```
$ date | grep Mon > /dev/null && echo "It's Monday"
```

is equivalent to

```
$ if date | grep Mon > /dev/null
> then
>     echo "It's Monday"
> fi
```

```
$ date | grep Mon > /dev/null || echo "It's NOT Monday"
```

is equivalent to

```
$ if ! date | grep Mon > /dev/null
> then
>     echo "It's NOT Monday"
> fi
```

THE `exit` COMMAND

- A shell script may contain the `exit` command. It is usually included as part of an `if` statement to cause an early termination of the script based on some logical test.
- `exit` may be given an argument, which will be the value returned by the shell script if it terminates by way of that `exit`.
- This allows a shell script to be called from another shell script that might need to determine whether the called script succeeded or not.
- If a script terminates by way of an `exit` with no return value specified as an argument, or if the script merely "falls off the bottom" (executes all the way to the last line) its return value will be that of the last command that executed before the `exit` was taken (or before it "falls off the bottom").
- If you are going to print any type of error message from within your script (as might be the case before you `exit` early with a non-zero return value) the standard output of `echo` should be redirected to the standard error. This ensures that if the user running your script has redirected standard error on the command line, your message will follow that redirection. Here's the syntax:

```
echo "This is an error..." >&2
```

COMMAND SUBSTITUTION: ``command``

- If the shell finds a pair of backquotes (``` ```) on a command line, the shell will attempt to perform **command substitution**.
- Do not confuse the backquotes with regular single quotes. The backquote (```) is often found on the same keycap as the tilde (`~`) character. The single quotes (`'`) are usually on the same keycap as the double quotes (`"`).
- The command line that appears within the backquotes is executed and any output it produces (its **standard output**) is inserted in place of the backquoted portion of the command line.
- Here is an example of inserting the output of the `date` command on an `echo` command line:

```
$ echo The date is `date`  
The date is Wed Dec 5 17:25:18 EST 2001
```

- The Korn shell and the "Bourne Again" shell (`ksh` and `bash`) also support this syntax, shown in **bold**:

```
$ echo The date is $(date)  
The date is Wed Dec 5 17:25:18 EST 2001
```

SIMPLE MATH: `expr` AND `$(())`

- Often, a shell script needs to perform arithmetic operations.
- Earlier, while examining the various forms of `test` and its associated operators, we saw that The Korn shell and the "Bourne Again" shell (`ksh` and `bash`) support the use of double parenthesis for performing integer comparisons. These double parenthesis may be preceded by a dollar sign (`$`) which allows them to perform arithmetic operations and substitute the result in place of the argument construct:

```
$ echo $((60-3)) # Or echo $(( 60 - 3 ))  
57
```

- The Bourne shell (`sh`) does not support this substitution. Instead, you may use the `expr` command (supported by all shells) to perform and return the result of an arithmetic operation. `expr` requires whitespace surrounding all of its operators and arguments:

```
$ expr 60 - 3  
57
```

- `expr` can be used for many other types of operations. See the documentation for `expr` for all the details.

SIMPLE MATH: `expr` AND `$(())` (continued)

- Here are two examples of storing the result of an arithmetic operation in a variable. The first will work with any shell (note that it uses backquotes for command substitution). The second is `ksh` and `bash` only. In either case, we are incrementing the value of `x` by 1:

```
$ x=0
$ x=`expr $x + 1`
$ echo $x
1
```

```
$ x=0
$ x=$(( $x+1 ))      # Or x=$(( $x + 1 ))
$ echo $x
1
```

- NOTE: Neither the `expr` command or the `$(())` form of arithmetic expression evaluation are capable of performing floating point math. All results are truncated to integer values. If you need to perform floating point (decimal) calculations, you can use the `awk` command described later on, or you can use the `bc` command that reads input from the standard input and prints its results to the standard output.
- Here is an example of deriving the floating point value of 12 divided by 7 with 4 digits to the right of the decimal point:

```
$ x=`echo "scale=4; 12/7" | bc`
```

or

```
$ x=$((echo "scale=4; 12/7" | bc))
$ echo $x
1.7142
```

EXERCISES

1. Change into the `Lab03` directory. Create a shell script called `WhereAreYou` that does the following:
 - Prompts the user to enter a user id (a login name).
 - Uses the `grep` command to examine the `/etc/passwd` file and determine whether or not the given user id exists anywhere in the file. If `grep` is unsuccessful, print a message using `echo` to indicate that the user id is not valid and use `exit` to leave the program with a `2` as the return value, otherwise print a message indicating that the user id is valid.
 - If `grep` DID find the user id in the `/etc/passwd` file, execute the `who` command and pipe its output into the `grep` command in an effort to determine whether or not the user id given is found anywhere in the output of `who`.
 - Prints one of two messages using `echo` to indicate whether or not the given user id is currently logged into the system. If the user id is not found in the output of `who`, after printing the appropriate message, `exit` the program with a `1` (one) as the return value.

EXERCISES (continued)

2. Create a shell script called `TimeToLeave` that calculates how many hours and minutes are left between the current time and 5:00 PM. For the arithmetic operations, you may use either the `expr` command or the `$(())` notation, depending on which shells are available on your system. Here are some guidelines:
- Executes the `date` command twice, once with the format argument `+%H` to obtain the current hour in the range of 00 through 23, and again with the format `+%M` to obtain the current minute in the range of 00 through 59. Save these values in two variables.
 - Use `expr` to subtract the current hour from 17 (5:00 PM in 24-hour format).
 - Initialize the remaining minutes to 0 (zero). If the current minute is greater than 0, use `expr` to subtract 1 from the remaining hours and subtract the current minute from 60 to obtain the remaining minutes.
 - Print the remaining hours and minutes until 5:00 PM using `echo`.

THIS PAGE IS INTENTIONALLY BLANK

EXERCISE SOLUTIONS

NOTE: These solutions also appear in your Lab03/solutions directory. These suggested solutions contain comments to help you understand what's being done. The comments are not necessary, they're just helpful. The code itself is printed in **bold**, while the comments are not. In addition, you should understand that your solutions don't have to look like these! The important thing is this: Do your scripts do what they're supposed to?

```
1.  #!/usr/bin/ksh
    # Solution to WhereAreYou:

    # Prompt for and save the user id:
    echo "Enter a user id: \c"
    read userid

    # Check to see if the user is in
    # the system's /etc/passwd file:
    if grep "^$userid:" /etc/passwd > /dev/null
    then
        echo "$userid is a valid user id."
    else
        echo "$userid is NOT a valid user id."
        exit 2 # Exiting with a failure code.
    fi

    # Check to see if the user id is logged on:
    if who | grep "^$userid" > /dev/null
    then
        echo "$userid is currently logged in."
    else
        echo "$userid is NOT currently logged in."
        exit 1 # Exiting with a different failure code.
    fi

    # Normal exit here "falling off the bottom
    # of the script".
```

EXERCISE SOLUTIONS (continued)

```
2.  #!/usr/bin/ksh
    # Solution to TimeToLeave:

    # Get and save the current hour (00-23):
hour=`date +%H`

    # Get and save the current minute (00-59):
minute=`date +%M`

    # Calculations:
hours_left=`expr 17 - $hour`
# Or
# hours_left=$(( 17 - $hour ))

minutes_left=0

if [ "$minute" -gt 0 ]
then
    hours_left=`expr $hours_left - 1`
# Or
# hours_left=$(( $hours_left - 1 ))

    minutes_left=`expr 60 - $minute`
# Or
# minutes_left=$(( 60 - $minute ))
fi

# NOTE the \ protecting the newline character:
echo "The time remaining until 5:00 PM is \
$hours_left hours and $minutes_left minutes."
```

THIS PAGE IS INTENTIONALLY BLANK