# CST8177

# bash Scripting

Chapters 13 and 14 in Quigley's
"UNIX Shells by Example"

# Control: The IF Statement

(Quigley pages 886 – 896)

The command is executed and its result (the same as **$?**) evaluated. If the result is zero, the **then** clause is executed; if it's not zero, it's skipped. This evaluation is then repeated in turn for each **elif** clause, if any.

If no **then** clause is executed, the statements in the optional **else** clause is executed.

```
if command
then
   statements
elif command
then
   statements
else
   statements
fi
```

# Numeric `if` example

- Full-form simple **`if`** statement

```
declare -i x=1
if ((x==1))
then
   echo x is $x
fi
```

- Or the compact form

```
if ((x==1)); then echo x is $x; fi
```

- Or the recommended form:

```
if ((x==1)); then
   echo x is $x
fi
```

# PDL for the `if` Control Statement

The PDL is really quite simple. Be sure to describe WHAT is happening and/or WHY it's being done.

```
IF command as PDL
     statements
ELSE IF command
     statements
ELSE
     statements
ENDIF
```

```
IF porridge burns
    PUT too hot
ELSE if porridge gluey
    PUT too cold
ELSE
    PUT just right
ENDIF
```

```
if ptemp > my_val; then
    echo too hot
elif ptemp < my_val
    echo too cold
else
    echo just right
fi
```

# Command `if` example

Using a command (no brackets needed):

```
if grep -q "string" file.txt; then
   echo \"string\" found in file.txt
elif grep -q "something" file.txt; then
   echo \"something\" found instead
else
   echo neither \"string\" nor \"something\"
fi
```

Be sure to get rid of any output to **stdout** (**grep**'s **-q**).

What PDL would have preceded this?

# Control: The CASE Statement

(Quigley pages 900 – 902)

The variable is compared with the values using the **shell** wildcards ( **? * [...]** ) , NOT regular expressions. All the statements are executed for the first matching value until the ending **;;**. If no value matches, then the default ***)** case is executed, if present.

```
case variable in
value1)
    statements
    ;;
value2)
   statements
    ;;
*)
   statements
    ;;
esac
```

# PDL for the case Control Statement

Some refuse to allow PDL for the **case** statement, in part because it's called different things in other languages (like **switch**) or because it's more like a long **if-elif** statement. I don't think that's reasonable, but it is necessary to describe it carefully. It's important to show the end of each clause as well as the end of the entire control construct.

```
CHOOSE from variable description
   CHOICE value1
      statements
   END CHOICE
   CHOICE value2
      statements
   END CHOICE
   DEFAULT CHOICE
      statements
   END CHOICE
END CHOOSE
```

# case Example

```
declare var="fred"
...
case "$var" in
   stuff)
      echo var was "stuff"
      ;;
   fred)
      echo var was "fred"
      ;;
   [fF]r?d)
      echo Fred, or something strange
      ;;
   *)
      echo default case
      ;;
esac
```

# Loops: do and done

A **do-done** pair is use to mark the start and end of every type of loop in **bash**. They frame a block of statements to be executed each time control passes through the loop.

Control over the loop is completely in the hands of whichever one of the control statements **for**, **while**, or **until** that precedes the **do-done** statement block:

```
for/while/until loop control
do
   statements
done
```

As with "**then**" in the **if** statement, we'll put the "**do**" after a semicolon in the control line. Similarly, the PDL will ignore the "**do**" and show the loop end as **END FOR**, **END WHILE**, or **END UNTIL**.

# Loops: The `for` Statement

(Quigley pages 903 – 907)

The **do-done** block is executed once for each word in the wordlist, assigning each word in turn to the variable. If "**in wordlist**" is omitted, the positional parameters **$@** from the caller starting at **$1** are used in its place.

```
for variable in wordlist; do
   statements
done
```

As stated above, the PDL will look like this:

```
FOR description of wordlist
   statements
END FOR
```

# **for Loop Example** (using **$()**)

```
ls
for x in $(ls); do
   cp $x $x'.backup'
done
ls
```

- The **ls** before the **for** statement shows:

**abc def ghi**

- The **ls** after the **for** statement shows:

**abc abc.backup def def.backup ghi ghi.backup**

# **for Loop Example** (using **$@**)

```
declare -i count=0
for x; do
    let count+=1
    echo Arg $count is $x
done
```

- If the command line had been invoked as:

  **./test-script apple banana cherry**

- Then the output from the **for** loop will be:

```
Arg 1 is apple
Arg 2 is banana
Arg 3 is cherry
```

# **Loops: The `while` and `until` statements** (Pages 907 – 912)

- The **do-done** loop is executed as long as command returns a value of zero (the same as true from **$?**).

```
while command; do
    statements
done
```

- **until** is the opposite of **while**, and executes as long as command returns a non-zero value (false or error from **$?**).

```
until command; do
statements
done
```

# Numeric loops: a common idiom

You will often see, or use yourself, a simple numeric loop. This example counts up, from 0 to $max, but others may count from 1 or may count down, from $max to 0 or 1. Note that counting often starts from 0.

```
declare -i i=0
...
while (( i < $max ))
do
   echo i has the value $i
   let i++
done
```

This example also uses "post-incrementing". That is, the adding of **1** to **i** is done at the bottom of the loop. You will also see it incremented (or decremented) at the top of the loop, depending on the purpose of the script.

# Numeric loops: using until

This is the same loop, using until instead of while. Notice that it's only the comparison that changes with the command.

```
declare -i i=0
...
until (( i >= $max ))
do
   echo i has the value $i
   let i++
done
```

# Note about loops and string compares

If you try to use a simple comparison like:

```
while (( $var==“fred” ))     # numerical
```

or

```
while [[ $var==“fred” ]]     # character
```

It sometimes, as in once in a while, won’t seem to work correctly.

Bash can be very picky at times when dealing with strings. I've heard this error reported but it's not very likely. If you should run into it, use:

```
while echo $var | grep -q "^fred$"
```

# I/O Redirection and Subshells for Loops (Pages 923 – 927)

The whole of a loop right down to the **done** statement can be treated as a unit for redirection and background processing, so that constructs like these are possible:

Pipe data into or out of a loop:

```
command | for do … done
while do … done | command
```

Redirect file output or input for a loop:

```
until do … done >> filename
while do … done < filename
```

Run a loop entirely in the background:

```
for do … done &
```

# The **break** and **continue** statements

(Pages 919 – 920)

- **break** exits from the current loop by executing the statement after the **done**. That is, it leaves from the bottom of the loop.

```
break
```

- **continue** send control to the command after the **do** statement. That is, it returns to the top of the loop.

```
continue
```

- Both have an optional number. When it's supplied, **break** breaks out of that many nested loops. **continue** returns to the top of the nth loop back.
- These are quite dangerous. Use with extreme caution. Better, **don't use them!**

```
break number
continue number
```

# The `null` statement (Pages 898)

The null statement **:** (colon) does nothing at all. Use it for empty **then** clauses, if you can't clearly reverse the test. Of course, the command-end character **;** (semicolon) does pretty much the same thing.

```
if some-test
then
  :
else
   # do something exciting
fi
```

# **The `select` statement** (Pages 912)

- Creating interactive text menus is much simplified by using the **`select`** loop and the **`PS3`** prompt.

  **`select variable in wordlist; do ... done`**

- Each item in wordlist (use quotes if an item includes blanks) is sent to **`stderr`** (not **`stdout`**) with a sequential number to its left, after a **`PS3`** prompt.
- The number selected by the user is stored in the **`REPLY`** variable (like **`read`**) and the corresponding word is placed in variable.
- Use a **`case`** or an **`if-elif`** to check the menu input.
- Because **`select`** is a loop control, the **`break`** command must used to exit from it (or **`exit`** to end the script). There is no ending condition as there is in **`for`**, **`while`**, and **`until`**.
- Choose your prompt and set **`PS3`** before your **`select`**.

# Sample PDL for select

```
SET prompt
SELECT from list of characters
   CHOOSE from value returned
      CHOICE Bart
          PUT message
      END CHOICE
      CHOICE Homer
          PUT message
      END CHOICE
      DEFAULT CHOICE
          BREAK out of loop
      END CHOICE
   END CHOOSE
END SELECT
```

# Implementation of select example

```
PS3="Who’s Your favorite Simpson? "
select x in "Bart" Homer 'Quit now'; do
   case $x in
      Bart)
         echo Eat my cow, man
         ;;
      Homer)
         echo "D'oh"
         ;;
      *)
         break # this choice exits loop
         ;;
   esac
done
```