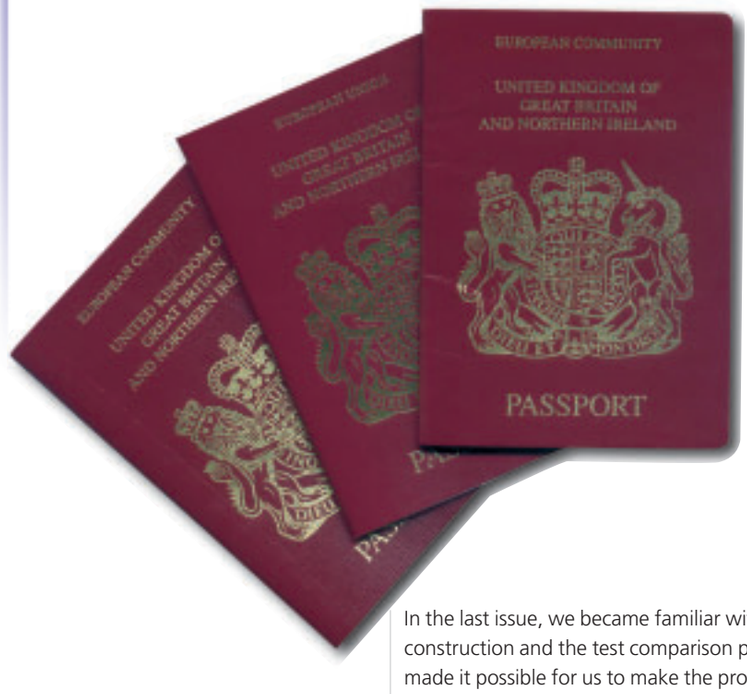
Control structures

CHECKPOINT CHARLIE

MIRKO DÖLLE



After the introduction to control structures and the presentation of simple comparison options in the last month's Programming Corner, this time we will be concerned with series comparisons, loops, keyboard inputs and small selection menus.

In the last issue, we became familiar with the *if* construction and the test comparison program. This made it possible for us to make the program sequence independent of external circumstances for the first time. Depending on the situation, other commands were executed. Bash knows additional control structures, which are especially necessary for larger comparisons and for multiple call-ups of individual commands (loops).

Parameter recognition

We shall begin with a small script. Like almost every other Linux program, our script will use the parameters -h or –>-help to output a brief explanation of the permissible options and then stop by itself. To do this, we use an if construct, covered last month:

```
#!/bin/bash
if [ "$1" = "-h" -o "$1" = "–>-help" ]; then
 echo "call up:"
 echo "  $0 [-h|–>-help]"
 echo "Parameter:"
 echo "  -h, –>-help: brief explanation"
fi
```

It becomes relatively fiddly and involved when we have to check several parameters and also wish to ignore upper and lower cases:

```
#!/bin/bash
if [ "$1" = "-h" -o "$1" = "-H" -o -z "${1#–>?
-[hH][eE][lL][pP]}" ]; then
 echo "-h"
elif [ "$1" = "-v" -o "$1" = "-V" ]; then
 echo "-v"
elif [ "$1" = "-q" -o "$1" = "-Q" ]; then
 echo "-q"
fi
```

The test on –>-help in the second line needs some explanation. So as not to have to test –>-help in all variants of upper and lower case, we use the pattern recognition from part 3 of our course. With ${1#–>-[hH][eE][lL][pP]} we search through the variable $1 for a string which begins with a double minus sign and contains an upper or lower case 'h', upper or lower case 'e' and so on. If there is a —>-help in $1 in any upper or lower case combination, it is removed – leaving a blank character string. This is where the test parameter -z comes into play. It supplies a true value if the following character string is empty – thus a version of –>-help has been found. Words which only begin with –>-help (for example –>-helper) fail the test, because the ending would be left over.

Simplification using *case*

As can be seen from the previous example: Large-scale parameter tests can hardly be conducted in

this manner. There is an urgent need for simplification. The method of parameter comparison is always the same: We check in each case whether the first parameter meets a certain condition. For such serial comparison, there is the *case* construction:

```
#!/bin/bash
case $1 in
 -h|-H|—>-[hH][eE][lL][pP])
   echo "-h"
   ;;
 -v|-V)
   echo "-v"
   ;;
 -q|-Q)
   echo "-q"
   ;;
esac
```

The *case* construct consists of the bracketing keywords *case* and *esac* (like the final *fi* for *if*, *esac* is also the reversal of the letters of *case*), a character string (here *$1*), which will be tested, and the individual blocks with the respective cases. These blocks begin with the pattern which is limited by a final round bracket, and end with a double semi-colon – in between are the instructions which are to be carried out for the respective case.

In our example we have three different cases, *-h*, *-v* and *-q*. The pattern of the first case consists of three parts, one of which must be true. The patterns themselves are practically identical to those from our *if* construct, although considerably clearer.

Apart from the square brackets which can be used to specify permitted characters or character ranges, there are also the wildcards *?* for any character and * for any sequence of characters. This makes it possible to distinguish the various network devices from each other:

```
case $device in
 eth*)
   echo "Ethernet"
   ;;
 ppp*)
   echo "Modem"
   ;;
 ippp*)
   echo "ISDN"
   ;;
 lo)
   echo "Loopback"
   ;;
 *)
   echo "Unknown"
esac
```

The last pattern, "*", applies to any character string – which is why this *case* construct in fact would always have to return "Unknown". But the cases are processed from top to bottom, and the only one to be executed is the first one that fits. All the others are ignored. So in the case of "ppp0" only "Modem" is output, but not "Unknown". The script is then continued, after dealing with the case, after the *esac* keyword.

Loops

With loops it is possible to have program segments executed many times, for example, to evaluate all command line parameters one after the other. The Bash knows three kinds of loop constructs: *for*, *while* and *until*. In principle the three loops are interchangeable with each other: Anything which can be solved using *for*, can in any case also be written using *while*. Nevertheless you should decide which loop construct is most appropriate for which problem.

for

The *for* loop is suitable for applications where a list of variables is laid down and has to be processed individually. This is practical for example for evaluating the command line parameters with our *case* construct:

```
#!/bin/bash
for P in $@; do
 case $P in
   -h|-H|—>-[hH][eE][lL][pP])
     echo "-h"
     ;;
   -v|-V)
     echo "-v"
     ;;
   -q|-Q)
     echo "-q"
     ;;
   *)
     echo "Unauthorised parameter $P"
     ;;
 esac
done
```

$@ provides a list of all command line parameters, which *for* then enters in sequence in the variable *P*, in order then to execute the body of the loop with our *case* construct.

If you're already familiar with other programming languages, you will be slightly surprised at the way the *for* loop works (in Perl for example it works in a completely different way). Usually, a start and an end value are stated, then the size of the increments by which the start value is to be raised. The loop is then run through until the end value is reached – which is usedto read in the values from 1 to 10 of a field, for instance.

This is not expressly provided for in the Bash, but we can remedy this by using the utility program *seq* from the *sh_utils* package (or *sh-utils*). *seq* supplies us with a number sequence from a specified starting value to an end value, there is the option of setting the size of the increments, and the

number format can also be changed. In order to output "Hello world" 10 times, we could use the following script, where the number of the respective run-through is placed in front in square brackets:

```
#!/bin/bash
for i in `seq 1 10`; do
 echo "[$i] Hello world"
done
```

Unfortunately there is no manual page for seq, but you will find relatively exhaustive help via the —>- help parameter. For home use, the invocations by means of seq Start End and seq Start Step size End are sufficient to cover most cases.

while

Certainly the most frequently used loop construction is *while*. In this case, the body of the loop is executed until the specified condition is true. One potential application is that of reading in any long field:

```
#!/bin/bash
i=1
while
 read -e -p "[$i]> ";
do
 field[$i]="$REPLY"
 : $[i+=1]
done
echo ${field[*]}
```

The condition in this case is an invocation of *read*, but, exactly as with the *if* construct, test or any other program can be used. If the return value of the program is 0, the condition is true, otherwise false.

The *read* command is new, with which we read inputs from the keyboard for the first time (exact standard input). The input is – unless otherwise specified – stored in the variable *REPLY*. The parameter *-e* activates the ReadLine extension, and you can edit the input line as usual with the cursor keys, and the tab completion of program names also works. The second parameter *-p "[$i]> "* defines the prompt, which is displayed at the beginning of the input line. In this case the element number is in the array, *$i* in square brackets with a smaller symbol and blank space following. Unlike *echo* there is no line break attached to the prompt, the cursor stays behind.

The rest of the script is quickly explained, *while* checks each time whether *read* is true – what exactly happens if something were to be entered. The input has put *read* into the variable *REPLY*, whose content we store in the body of the loop as element number *$i* of our array *field*. Then *i* is increased, so as not to overwrite the stored values, and the loop starts again from the beginning.

It is only when, instead of a value, [Ctrl+D] is pressed, the return value of *read* does not equal 0, the condition is thus not met. The loop ends, and the next instruction after the *done* belonging to the loop is executed. In our case it is the output of all elements of our array *field*, which we achieve by specifying a start in place of the element number.

Table 1: *read* parameters

-a *Field*	Allows the input of several values separated by spaces. The values are stored in the array *Field* , incrementing from element 0 on
-e	Activates the ReadLine support. This makes it possible for example to edit the input line with cursor keys or tab completion
-r	Deactivates the backslash-enter special treatment. Normally it is possible to continue an entry on the next line by means of a backslash at the end of a line, without the line break having any effect
-p *Prompt*	Replaces the standard input challenge with the character string *Prompt*, without attaching a line break.
Name	Stores the input directly in the variable *Name* and not, as usual, in *REPLY*

Program simplification

The example just shown is still very exhaustive and even relatively complicated as it is written. There is a much shorter and faster way:

```
#!/bin/bash
while
 read -e -p "[$[i+=1]]> " field[$i];
do
 :
done
echo ${field[*]}
```

The most unusual thing, at first glance, is the empty body of the loop, containing only the colon as zero function. But this is necessary, since the body cannot be empty. There is no need to restore *REPLY* in the field element, because by specifying the variable name *field[$i]* as last parameter of *read*, the values entered are stored immediately in the array and not in *REPLY* first. Incrementing the variable *i* by 1 is also transferred into the condition and in addition the initialisation has been done away with by the 1. The crux of the matter is that new variables are empty by default, but have the arithmetical value 0. The instruction *$[i+=1]* first increments the variable by 1 and then delivers the value. So we begin as before at element number 1.

until

until incidentally does the same as *while*, except that the loop is executed as long as the condition is false – otherwise there is no difference. The following example is something you will be familiar with from the presentation of the *while* loop, the condition has been inverted by means of exclamation marks – true becomes false and vice versa.

```
#!/bin/bash
i=1
until
 ! read -e -p "[$i]> ";
do
 field[$i]="$REPLY"
 : $[i+=1]
done
echo ${field[*]}
```

In Listing 1 you will find the rough draft of a script, which apart from -h and –>-help also understands the parameters -q and –>-quiet respective as well as -v which means the same as –>-verbose. -q and -v are often used either to suppress all outputs except for serious error messages and/or to comment all actions. The default setting is the verbose mode via the variable QuietMode, by setting this to 1 in the second line. Even if at first sight this looks nonsensical: Since 0 is true, we set QuietMode with 1 to false.

Selection menus

The *select* construction offers an option which is frequently underestimated. This makes it possible to construct complex selection menus. Here is a simple example for choosing between apples, pears and plums:

```
#!/bin/bash
Field=("apples" "pears" "plums" "end")
select fruits in ${field[*]}; do
 case $REPLY in
  ${#field[*]})
    return
    ;;
  *)
    echo "$Fruits"
    ;;
 esac
done
```

The content of our array is hidden behind *${field[*]}*, in this case four elements, which *select* numbers in sequence and outputs one after another:

```
1) Apples
2) Pears
3) Plums
4) End
#?
```

Now *select* queries the number of the desired action, for which *read* is implicitly used. The result of the selection is as usual available later in the variable *REPLY*. In addition *select* copies the corresponding element in our array into the variable *P* and executes the body. Once this has been processed, *select* shows the selection again.

The rest of the procedure in the body is defined by means of *case*. The first pattern looks fairly unusual at first, but *${#field[*]}* is concealing only the number of elements contained – which is the same as the number of the last entry. This allows us to reliably recognise when the user has selected the last entry, without having to know its actual number or name – thus the selection can be expanded at will, as long as the last entry stands for "exit menu".

Listing 1

```
#!/bin/bash
QuietMode=1
for P in $@; do
 case $P in
   -h|-H|–>-[hH][eE][lL][pP])
     echo "Command:"
     echo "  $0 [-h|–>-help]|[-q|–>-quiet]|[-v|–>-verbose]"
     echo "Parameter:"
     echo "  -h, –>-help:    This brief explanation"
     echo "  -q, –>-quiet:   Only report serious errors"
     echo "  -v, –>-verbose: Extensive messages"
     exit
     ;;
   -v|-V|–>-[vV][eE][rR][bB][oO][sS][eE])
     QuietMode=1
     ;;
   -q|-Q|–>-[qQ][uU][iI][eE][tT])
     QuietMode=0
     ;;
   *)
     echo "error: Unknown parameter $i"
     exit
     ;;
 esac
done
echo $QuietMode
```

In order to leave *select*, you must either press [Ctrl+D] or, as shown in the example, call up *return* or *break*. To this extent, *select* also differs from all other loops – it has neither a condition nor a list, after the processing of which the loop ends.

Unlike *read*, you cannot give *select* the prompt which is to be output as parameter – the standard return prompt from the variable *PS3* is used, which you can adapt manually:

```
#!/bin/bash
field=("apples" "pears" "plums" "end")
PS3="Which fruits? > "
select fruits in ${field[*]}; do
 case $REPLY in
  ${#field[*]})
    return
    ;;
  *)
    echo "$fruits"
    ;;
 esac
done
```

That ends the fifth part of Programming Corner. part 6 will be on the structuring and modularisation of scripts by means of functions and modules. Using a small management program we will then recap all the previous lessons in part 7 and show the potential applications of the individual commands and constructs. ■