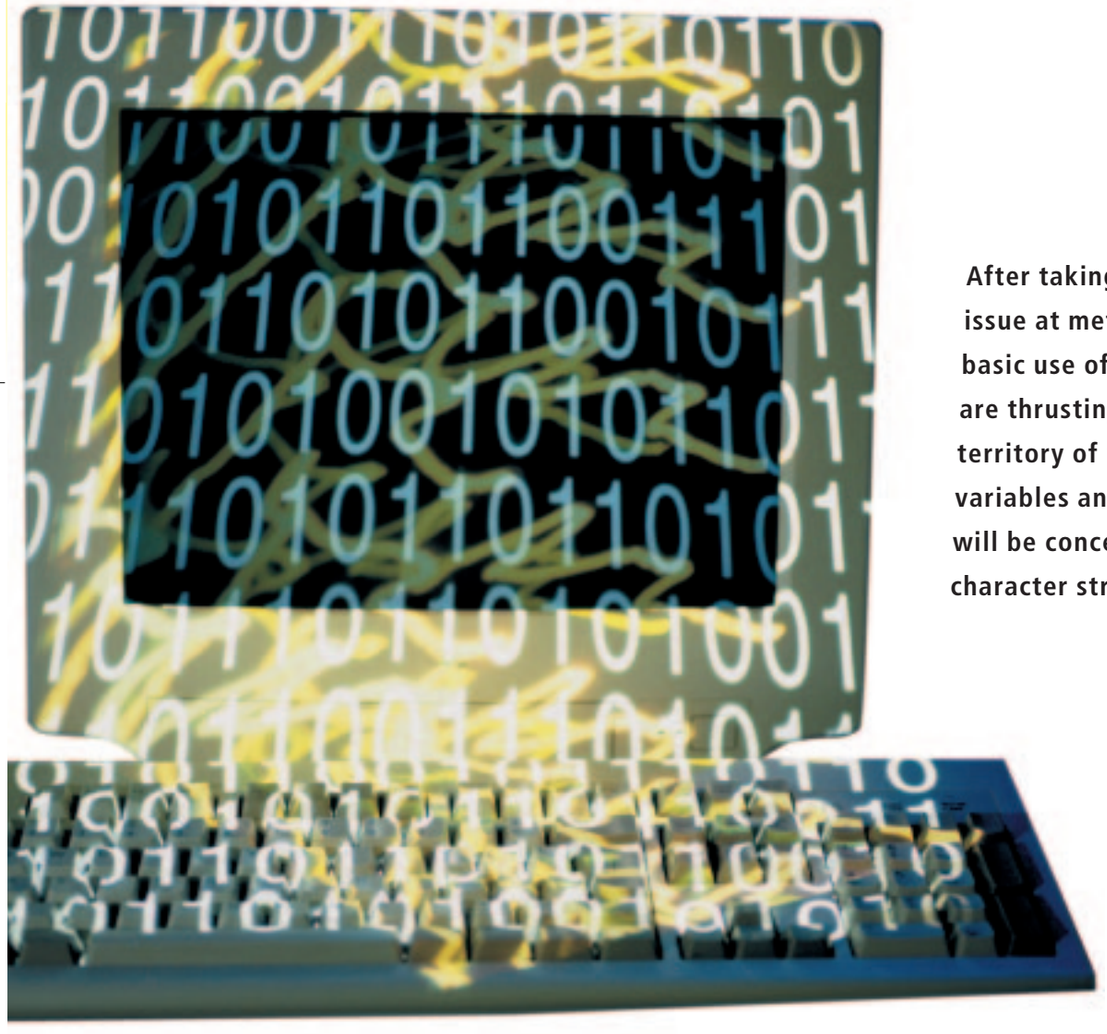
094pcorner.qxd 31.01.2001 14:27 Uhr Seite 94

Part 2: Principles of Bash

ARRAY OF LIGHT

MIRKO DÖLLE



After taking a look in the last issue at metacharacters and the basic use of variables, this time we are thrusting forward into the territory of multidimensional variables and towards the end we will be concerned with processing character strings.

This time we are going to start with variable fields, the so-called arrays. It may sound very complicated but in reality it's very simple to use. A one dimensional array is nothing but a series of variables one after the other, as if one had a little box of broad strips of squared paper – in each box one can write a number. Since in these array-variables there is no longer a single value, we have to say each time which box we mean. To do this we attach to the name of our fields in square brackets the number of the box, which is to be used:

```
V[1]=Hello
V[2]=world!
```

094pcorner.qxd 31.01.2001 14:27 Uhr Seite 95

In the first line we write in box number 1 *Hello* and in box number 2 *world!*. As so often in the computer world, Bash also starts to count at zero. So before "Hello" we still have one box free:

```
V[0]="We say:"
```

For the element zero incidentally the explicit designation over "[0]" can be left out, both in assignments and in the output, as the following output shows:

```
echo $V ${V[1]} ${V[2]}
We say: Hello world!
```

And this is the snag with arrays: to reach an element, i.e. the content of a box, we have to use square brackets – otherwise Bash would think "$V[1]" was "$V" – thus "$V[0]" – and then "[1]" instead of the number of element number one in our array.

So let's just take a look at what is actually in our array. To do this we need the command *typeset*, with which among other things one can query the status and content of a variable:

```
typeset -p V
declare -a V='([0]="We say:" [1]="Hello" ➥
[2]="world!")'
```

The output from *typeset* corresponds to what one would have to enter to roll in the array new from the ground up. The new part is the command *declare -a*, with which one can log variables. "-a" explicitly defines that *V* is an array, following which the values of the elements 0, 1 and 2 are entered. *declare* has other parameters apart from "-a", which we will look at as necessary. As so often happens, we can also do without the *declare* in this case, by simply writing:

```
V=([0]="We say:" [1]="Hello" [2]="world!")
```

At this point, just a word about programming in general. A programming language serves to give the computer instructions in forms which are legible and comprehensible for humans. One could also feed the computer direct with processor commands, but then one would have serious problems correcting errors later or installing new functions – after a certain time, one would no longer understand one's own program. The last stop on this line is often the waste paper basket, together with a complete redesign of the program.

This is why it is very worthwhile to use *declare* to declare more complex structures, arrays for example. An outsider is then much more likely to understand the program. But that's not enough.

You really must get used to documenting your programs, regardless of whether you are compiling them for C or Bash. The middle way between the Spartan copyright annotation and a comment on every line is, as usual, the best. An output line, in which you concisely report an error, is something you don't need to document. You may assume that a reader has complete mastery of Bash. But if you start to process data with external programs and at the same time install interlocks or even tricks for faster servicing, this point obviously really must be documented with more than one line. I will, when we come in the next instalment to corresponding examples, go into this again.

But back to the arrays. Let's just assume we had gaps in our field, as in the following example:

```
N="The"
N[3]="house"
N[6]="."
```

On our strip of squared paper we would occupy boxes 0, 3 and 6. Bash manages its memory better, though, and there are no gaps there:

```
typeset -p N
declare -a N='([0]="The" [3]="house" [6]=".")'
```

Our three entries are simply stored one after the other, plus Bash remembers their element number. In this way we can at any time fill in the as yet unused numbers in the gap:

```
N[1]="is"
N[2]="the one"
N[4]="owned by"
N[5]="Nikolaus"
typeset -p N declare -a N='([0]="That" ➥
[1]="is" [2]="the" [3]="house" [4]="owned by" ➥
[5]="Nikolaus" [6]=".")'
```

Finally it should be mentioned that array-elements can be used at any point where a normal (scalar) variable could be placed, but in most cases one simply has to use the notation with the curly brackets.

Special variables

Bash has access to a whole range of special variables, such as for example the parameters for a program start. Here are just the most important ones.

The variables *$0*, *$1*, *$2* etc. are the parameters which were provided by the user when invoking a Bash script (or a function, but more on that later). You can work with these in exactly the same way as with all other variables, except that you cannot assign values to them. This is because variable names cannot start with a figure. The variable *$0* is always set. This is where the program name, as used to call it up, is found. The total number of parameters can be polled with *$#*. Take note that *$0* is not included in this count, if *$#* thus supplies "4", this means that there is *$1* to *$4* plus *$0* in addition.

Let's just take a look at the example "myecho" in the following listing:

```
#!/bin/bash
echo [$#]: $1 $2 $3 $4 $5 $6 $7 $8 $9 ${10} ➥
${11} ${12}
```

094pcorner.qxd 31.01.2001 14:28 Uhr Seite 96

The curly brackets from parameter ten on are necessary, by the way, as otherwise Bash, as will be familiar from arrays, will first insert *$1* and then add on a zero, etc. Simply write the little program in a text editor, for example *kedit*, *mcedit* or even *emacs*, and save it under the name *myecho*. Then you have to make the file executable with the command *chmod a+x myecho*, and call it up twice:

```
./myecho 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
[15]: 1 2 3 4 5 6 7 8 9 10 11 12
./myecho "1 2 3 4 5 6 7 8 9 10 11 12 13 14 15"
[1]: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

The different result from the two calls is due to the quote marks. While on the first occasion we have, according to *$#*, 15 parameters, with the second call there is only one involved. Bash interprets spaces as separators between two parameters in program calls; consequently the 15 numbers separated by blanks are 15 parameters. As already outlined in the first part, we must either quote parameters containing blanks in full or leave out the blanks. In the second call Bash thus now has only one parameter, consisting of all 15 numbers separated by blanks, while *$2* to *${12}* from our program remain unused.

To be able to output any desired number of parameters, we would have to go through and individually output all parameter-variables from *$1* to the end, for example in a loop. The two special variables *$** and *$@* save us this task, here for example is our altered *myecho*:

```
#!/bin/bash
echo [$#]: $*
```

The difference between *$** and *$@* only becomes clear when both are placed in quotes. Then when there are four parameters *"$*"* turns into *"$1 $2 $3 $4"*, while *"$@"* turns into *"$1" "$2" "$3" "$4"*. At first glance the difference does not appear to be of any significance. But that's due to the internal Bash variable *IFS* being pre-set. The first character of this variable is used with *"$*"* to subdivide the individual parameters. Normally *IFS* contains three characters, namely the Blank space, the Tabulator and Enter. The following example shows the difference, by placing a comma before the standard character in IFS:

```
#!/bin/bash
IFS=",$IFS"
echo [$#]: "$*"
echo [$#]: "$@"
```

The third line lists for us all numbers separated by a comma, while the fourth is still separated by blanks.

On the other hand *$@* is not superfluous, either, and for this example we shall take a new program with the name *whichfiles*:

```
#!/bin/bash
ls -l "$@"
echo ––
ls -l "$*"
echo ––
ls -l $@
echo ––
ls -l $*
```

Please remember to also make the program executable with *chmod a+x whichfiles*. All we need now is two files, one of them with spaces in the name. Next we call up *whichfiles* and specify – careful with the blank spaces – both files as parameter:

```
echo "Hello world" > "hello world.txt"
echo "That is Nikolaus's house" > nikolaus.txt
./whichfiles "hello world.txt" nikolaus.txt
```

The result requires a bit more explanation. In this case we have placed all four notations of *$** and *$@* one after the other, the *echo* instructions are only used as separators to give a better overview. The first *ls* call correctly showed us both files, *hello world.txt* and *nikolaus.txt*. As expected *$@* turned into *hello world.txt* and *nikolaus.txt*, *ls* thus received two parameters and displayed both files. The second *ls* rightly complained at being unable to find any hello world.txt nikolaus.txt. The reason is the property of *$**, of supplying all parameters separated by a character in quotes. Accordingly *ls* also received only one parameter, i.e. hello world.txt nikolaus.txt. Calls three and four provide the same result; here both parameters are given to *ls* separated by a space. Because of the second space in *hello world.txt* there were three parameters for *ls*, *hello* and *world.txt* thus correctly gave rise to a complaint, as neither of these files exist.

Finally the variables *$?* and *$!* should be mentioned, which serve to query program response

Special variables in Bash

$*	All parameters assigned to the program, separated by blanks. $*=$1 $2 $3 $4 ...
"$*"	All parameters assigned to the program, in quotes, separated by the 1st character (c) of the variable IFS. "$*"="$1c$2c$3c$4" ...
$@	All parameters assigned to the program, separated by blanks. $@=$1 $2 $3 $4 ...
"$@"	All parameters assigned to the program, individually enclosed in quotes and separated by blanks. "$@"="$1" "$2" "$3" "$4" ...
$#	Number of assigned parameters
$?	Response value of the last command
$$	Process-ID (PID) of the current program
$!	Process-ID (PID) of the last program started in the background
$_	Last parameter of the last program called up

094pcorner.qxd 31.01.2001 14:28 Uhr Seite 97

values, the so-called *exit status*. At present they are not yet important to us, but a short description can be found in the table "Special variables for Bash".

String processing

Most script languages, as well as Bash, are principally concerned with processing character strings, also referred to simply as strings. This includes a series of letters, numbers, special and control characters, for example the chapter of a book. It would be wrong only to take into account letters, numbers and any special symbols in character strings, because a string can frequently be several lines in length and contain other formatting symbols such as tabulators or a page break. This clarification is important – we must always be aware that in such a string variable there could always be a whole novel or just one file.

The assignment and output of a character string has been shown by many examples. Next we come to the possibilities for finding out something about the text hidden in variables and manipulating it. Contrary to Version 1.1 of Bash, we now have access to a vast range of functions, but let's start with something very trite.

How do you find out that a variable is empty? Well, one possibility would be to compare the content with an empty string (which means we are concerned with the control structures), or else simply determine the length of the string:

```
a=""
echo ${#a}
0
a="four"
echo ${#a}
4
```

The quotes in the assignment can be left out, even if it would look somewhat unusual in the first line.

We can even – without first having introduced control structures – react to an unspecified parameter with an error message:

```
#!/bin/bash
a="Hello"
: ${a:?'yes'}
echo 'no'
```

Save this little program under *is-a-empty* and call it up after you have made it executable with *chmod*. The program answers correctly with no, which is hardly surprising, since after all, we did output it with *echo*. Now delete *Hello* from the second line, so that *a* is now empty, and call up the program once more:

```
a: yes
```

It is obvious that our *echo no* from the last line of the program has not been executed this time. That's the result from the third line. We know of the colon-command from the last instalment of Programming corner: it does nothing at all. The parameters after colon, however, are still taken into account and this is where the evaluation is hidden. The instruction *${variable:?errormessage}* first checks whether *variable* is empty or does not even exist. If so, the error message after the question mark is issued and the program is interrupted. That was why the *echo* from the fourth line did not even get a look in. When the error message is output, Bash has, in the usual way, placed the program name and the location at which the error arose, in front.

For the next two instructions we shall modify our script *whichfiles* first:

```
#!/bin/bash
ls -l ${1:-$HOME}
```

What happens now is that we get a list of either the specified directory, or else the content of our home directory *$HOME*:

```
./whichfiles /bin
arch
bash
cat
...
./whichfiles
dead.letter
mail ...
```

The instruction *${variable:-string}* has the effect of inserting *string* where there are empty or non-existent *variables*. In our case the content of the variable *$HOME*, otherwise the content of *$variable*. We shall check the first parameter in the program. If we have specified a directory, *$1* is filled and the instruction delivers the content. If not, *$HOME* is inserted.

Almost the same effect is produced by *${variable:=string}*, but here *string* is additionally assigned to the *variable*, if it is empty or does not exist:

```
#!/bin/bash
directory="$1"
: ${directory:=$HOME}
echo "show $directory"
ls -l "$directory"
```

${variable:+string} has a similar effect to the previous instructions. The difference consists in that *string* is always inserted when *variable* contains something. If it is empty, nothing is inserted. This instruction is very seldom used in practice, which is why I have also been unable to find a useful example of it.

In the next instalment of Programming corner we shall be taking a look at the initiation of part-strings and Search/Replace with regular expressions in Bash. For now all that is left is for me to wish you a good start to the new millennium which now lies before us. ■

Terms

Character string, string: *A series of letters, numbers, special and control characters. Character strings can certainly contain several lines, and the content can also consist of a program or image file.*
Field, array: *Consists of several* ***Elements****, which are addressed via numbers. In one-dimensional arrays, too, only one number is necessary for the selection of an element, for two-dimensional arrays two, etc.*
Element of an array: *This is addressed via its position. Elements can be used like regular variables; values can be stored and called up.*
Response value, exit status: *Value sent back to the enquirer when a program ends. The response is guaranteed by the system. Not to be confused with messages on the screen. If the program has been successfully executed, the value 0 is usually returned, if errors or problems arose, the value is greater than 0. One can often determine the error on the basis of the response values; explanations on their meaning can usually be found in the program documentation.*

■