# Bash - Array-Technik


Auch jene, die schon über 20 Jahre mit der Bash arbeiten, kennen oder beachten oft die Bash-Array-Funktionen gar nicht. Diese Übungen zeigen, wie man Arrays auch in der Bash verwenden kann, so wie diese in anderen Skript-Sprachen wie z. B. In Perl ebenfalls angewandt werden. Arrays ermöglichen es, eine geordnete Folge von Werten eines bestimmten Typs zu speichern und zu bearbeiten. Allerdings erlaubt die Bash nur eindimensionale Arrays. Dabei werden zwei grundlegende Typen von Arrays unterschieden.

*Indizierte Arrays* verwenden positive Integer-Zahlen als Index. Sie sind meist *sparse*, die Indizes sind nicht unbedingt zusammenhängend. Wie in der Hochsprache „C“ und vielen anderen Programmiersprachen beginnen die ganzzahligen Array-Indizes bei 0. Indizierte Arrays wurden zuerst bei der Bourne-ähnlichen Shell, der *ksh* eingeführt. Indizierte Arrays tragen immer das **Attribut `-a`**.

Assoziative Arrays (manchmal als ***Hash*** bezeichnet) verwenden beliebige nichtleere Zeichenketten als Index. Assoziative Arrays sind immer ungeordnet, sie assoziieren nur Index-Wert-Paare. Wenn Sie mehrere Werte aus dem Array abrufen, können Sie sich nicht darauf verlassen, dass diese in derselben Reihenfolge wiedergegeben werden, in der sie eingefügt wurden. Assoziative Arrays tragen immer das ***Attribut `-A`***. Im Gegensatz zu indizierten Arrays müssen sie immer ausdrücklich deklariert werden.

## *Indizierte Arrays*

Ein indiziertes Array (Feld, Vektor, Reihung) ermöglicht die Verarbeitung mehrerer gleichartiger Elemente, wobei auf jedes Element über seinen Index eindeutig zugegriffen werden kann. Oft werden Arrays in Schleifen verwendet, so dass man nicht auf eine Reihe von einzelnen Variablen zurückgreifen muss.

Die folgenden Möglichkeiten versehen eine Variable mit Array-Attribut, wobei beim indizierten Array keine Deklaration notwendig ist. Diese erfolgt meist implizit durch die Wertzuweisung:

- **`ARRAY=()`** deklariert ein indiziertes Array namens »ARRAY« und initialisiert es als leer. Dies kann auch zum Leeren eines bestehenden Arrays verwendet werden.
- **`ARRAY[0]=xxx`** besetzt das erste Element eines indizierten Arrays (Wertzuweisung). Wenn noch kein Array namens »ARRAY« existiert, wird es erzeugt.
- **`declare -a ARRAY`** deklariert ein indiziertes Array namens »ARRAY«. Ein bereits vorhandenes Array wird dadurch nicht initialisiert.

Wenn Sie einer Array-Komponenten einen Wert zuweisen wollen, erfolgt dies genauso wie bei den normalen Shell-Variablen. Sie geben lediglich noch den Feldindex eingeschlossen in eckige Klammern an. Insofern unterscheidet sich die Bash nicht von C, Perl, Python oder anderen Programmiersprachen. Sie müssen lediglich daran denken, dass rechts und links vom Gleichheitszeichen kein Leerzeichen stehen darf. Im folgenden Beispiel erfolgt das Belegen des dritten(!) Array-Feldes mit dem String »zwo«:

**`array[2]=zwo`** oder **`array[2]="zwo"`**

Was geschieht in diesem Fall mit den Komponenten `array[0]` und `array[1]`?
Bei der Bash-Programmierung dürfen Arrays Lücken enthalten (wie z.B. auch in Perl oder anderen Interpretersprachen). Im Unterschied zu anderen Programmiersprachen müssen Sie in der Bash das Array auch nicht vor der Verwendung deklarieren. Sie sollten es jedoch trotzdem zu tun, denn einerseits kann man damit angeben, dass die Werte innerhalb eines Arrays z. B. als Integer gültig sein sollen (**`typeset -i array`**) und andererseits können Sie ein Array mit Schreibschutz versehen (**`typeset -r array`**).

Häufig will man beim Anlegen eines Arrays dieses mit Werten initialisieren. In der Bash erfolgt dies durch Klammern der Werte:

```
array=( Merkur Venus Erde Mars Jupiter Saturn )
```

Bei der Zuweisung beginnt der Feldindex automatisch bei 0, also enthält `array[0]` den Wert »Merkur«. Aber ist es auch möglich, eine Zuweisungsliste an einer anderen Position zu beginnen, indem der Startindex angegeben wird:

```
array=([2]=Pluto Eris ... )
```

Diese Methode ist jedoch nicht geeignet, an ein Array neue Elemente anzuhängen. Ein existierendes Array wird immer komplett überschrieben. Die Anzahl der Elemente, die Sie einem Array übergeben können, ist bei der Bash beliebig. Als Trennzeichen zwischen den einzelnen Elementen dient das Leerzeichen.

Auch aus einer Datei lässt sich ein Array erzeugen. Es wird der Dateiinhalt per `cat`-Kommando und **`$(...)`** auf die Kommandozeile gehoben und er landet damit in der Initialisierungsliste:

```
array=($(cat "datendatei"))
```

Der Zugriff auf die einzelnen Komponenten eines Arrays erinnert an C oder andere Sprachen. Er erfolgt mittels

```
${array[index]}
```

Einziger Unterschied zu anderen Sprachen: Für die Referenz auf den Arrayinhalt müssen geschweifte Klammern verwendet werden, da die eckigen Klammern ja schon Metazeichen der Bash sind (Bedingung analog dem `test`-Kommando) und sonst eine Expansion auf Shell-Ebene versucht würde.

Alle Elemente eines Arrays können Sie, ähnlich wie bei den Kommandozeilenparametern, folgendermaßen ausgeben lassen:

```
echo ${array[*]}
echo ${array[@]}
```

Der Unterschied ist der gleiche wie bei den Kommandozeilenparametern ($* und "$"). Ohne Anführungszeichen expandieren beide Ausdrücke gleich, mit Anführungszeichen expandiert **`${array[*]}`** zu allen Elementen in einem, während **`${array[]}`** zu den einzelnen Elementen in Anführungszeichen expandiert. Das ist insbesondere bei for-Schleifen wichtig, die über die einzelnen Elemente iterieren.

Mit **`${!array[*]}`** lassen sich alle Indizes des Arrays auflisten. Dieses Feature ist erst später hinzugekommen. Diese Liste ist interessant, wenn das Array Lücken bei den Indizes aufweist. Im folgenden Beispiel werden Index und Inhalt aufgelistet:

```
01: for IND in ${!array[@]}
02: do
03:   echo "$IND ${array[$IND]}"
04: done
```

Die Anzahl der belegten Elemente im Array erhalten Sie mit (auch diese Syntax ähnelt der Kommandozeile bzw. der Shell-Variablen **$#**):

```
echo ${#array[*]}
```

Wollen Sie feststellen, wie lang eine Array-Komponente ist, so gehen Sie genauso vor wie beim Ermitteln der Anzahl belegter Elemente im Array, nur dass Sie anstelle des Sternchens den entsprechenden Feldindex verwenden:

```
echo ${#array[1]}
```

### *Löschen, Kopieren und Konkatenieren*

Zum Löschen eines Arrays oder einer Komponente wird wie bei den Shell-Variablen vorgegangen. Das komplette Array löschen Sie mit `unset array`. Einzelne Elemente können Sie durch Angabe des Index löschen: `unset array[index]`. Das funktioniert aber nicht immer, zum Beispiel:

```
array=(Merkur Venus Erde Mars Jupiter Saturn Uranus Neptun)
INDEX=4
echo {#array[@]}
8
echo ${array[$INDEX]}
Jupiter
unset ${array[$INDEX]}
echo ${#array[@]}
8
# Oha! Das sollte aber nun 7 sein, weil Jupiter ja geloescht wurde. Mal sehen:
echo ${array[$INDEX]}
Jupiter
# Ooops! Den gibt es ja immer noch!
```

Der Fehler liegt darin, dass in der Bash ein Array als sogenannter Hash gespeichert wird (»sparse«). Sie sollten sich also nicht darauf verlassen, dass eine Komponente nach dem `unset` tatsächlich weg ist.

Zum Kopieren eines kompletten Arrays können Sie entweder eine Schleife programmieren, in der jede einzelne Komponente des einen Arrays dem anderen Array zugewiesen wird. Es geht aber auch mit weniger Aufwand. Sie kombinieren das Auflisten aller Elemente des einen Arrays mit dem Zuweisen einer Liste mit Werten an ein zweites Array:

```
array_neu=( ${array[@]} )
```

Auch hierzu ein Beispiel:

```
array=(Merkur Venus Erde Mars Jupiter Saturn)

# Kontrollausgabe
echo ${array[@]}
Merkur Venus Erde Mars Jupiter Saturn

# array auf yarra kopieren
yarra=( ${array[@]} )

# Kontrollausgabe
echo ${yarra[@]}
Merkur Venus Erde Mars Jupiter Saturn
```

Das Aneinanderhängen zweier Arrays geht genauso wie das Konkatenieren von Strings. Es wird einfach eine Initialisierungsliste aus beiden Arrays gebildet, die dann einem neuen Array zugewiesen wird:

```
Gesamt=("${Array1[@]}" "${Array2[@]}")
```

## *Zwei Praxisbeispiele*

Eine Liste der Dateinamen des aktuellen Verzeichnisses bekommen Sie übrigens mit der Zuweisung:

```
Dateien=(*)
```

Zum Abschluss eine praktische Nutzanwendung für Arrays. Bei Raspberry Pi (meine derzeit liebste Spielmaschine) müssen die GPIO-Ports für die parallele Ein- und Ausgabe zu Beginn initialisiert werden. Sind es zahlreiche Ports, macht man das am besten in einer Schleife. Nur steckt die Schleife meist in den Tiefen des Skripts und man muss für das Ändern oder Hinzufügen von Ports erst einmal suchen. Viel wartungsfreundlicher ist es, die Portzuordnung am Anfang des Skripts (oder sogar in einer Konfigurationsdatei) vorzunehmen. Das sieht dann beispielsweise folgendermaßen aus:

```
# Port-Zuordnung zu den LEDs
GPIO=(27 22 19 17 13 6 5)
```

Die Initialisierung der Ports erfolgt dann in einer Schleife über die Array-Komponenten, was in Zeile 2 geschieht. Nach Expansion durch die Shell lautet die Zeile »`for Port in 27 22 19 17 13 6 5`«, also eine ganz normale Schleifenanweisung. Die Zeilen 4 bis 11 sorgen für die Initialisierung der GPIO-Ports, der genaue Inhalt ist an dieser Stelle nicht so wichtig, hier ist nur die Anwendung der Laufvariablen `Port` von Belang.

```
01: # Ports initialisieren, wenn noch nicht erfolgt
02: for Port in ${GPIO[@]}
03: do
04:   if [ ! -f /sys/class/gpio/gpio${Port}/direction ] ; then
05:     echo "Initialisiere GPIO $Port"
06:     echo "$Port" > /sys/class/gpio/unexport 2> /dev/null
07:     echo "$Port" > /sys/class/gpio/export
08:     echo "out" >/sys/class/gpio/gpio${Port}/direction
09:     echo "0" >/sys/class/gpio/gpio${Port}/value
10:     chmod 666 /sys/class/gpio/gpio${Port}/direction
11:     chmod 666 /sys/class/gpio/gpio${Port}/value
12:   fi
13: done
```

Übrigens könnten Sie mit dem Befehl **`PARAM=( $@ )`** auch alle Kommandozeilen-Parameter in einem Array speichern und dann wahlfrei darauf zugreifen. Das ist manchmal einfacher als andere Methoden des Zugriffs auf die Parameter. Um das Beispiel oben weiterzuspinnen, soll auf der Kommandozeile beim Aufruf des folgenden Skripts für jede LED »0«oder »1« angegeben werden und diese Parameter der Reihe nach auf den oben definierten GPIP-Ports ausgegeben werden, also der erste Parameter auf Port 27, der zweite auf Port 22 usw. Als Laufvariable dient der Index des Parameterarrays (Zeile 2). In der Schleife wird dann der I-te Parameter auf den I-ten GPIO-Port ausgegeben (Zeile 4).

```
01: PARAM=( $@ )
02: for I in ${!PARAM[@]}
03: do
04:   echo ${PARAM[$I]} > /sys/class/gpio/gpio${GPIO[$I]}/value
05: done
```

## *Assoziative Arrays*

Assoziative Arrays gibt es in der Bash ab Version 4. Ein assoziatives Array kann man sich als Verknüpfung von zwei Arrays vorstellen, wobei eines die Daten enthält und das andere die Schlüssel, welche die einzelnen Elemente des Datenarrays indizieren. Assoziative Arrays müssen explizit deklariert werden.
Die Anweisung **`declare -A ARRAY`** deklariert ein assoziatives Array namens »ARRAY«. Dies ist die einzige Möglichkeit, assoziative Arrays zu erstellen.

Die Zuweisung eines Wertes erfolgt genauso wie bei den indizierten Array, nur dass hier ***kein Skalar*** als Index verwendet wird, ***sondern eine Zeichenkette (Key).*** Zum Beispiel:

```
declare -A namen
namen[Donald]=Duck
namen[Daniel]=Düsentrieb
namen[Gustav]=Gans
```

Im Key dürfen auch Leerzeichen oder andere »exotische« Zeichen auftauchen. Dann muss der Key aber in Anführungszeichen gesetzt werden, beispielsweise:

```
namen["Onkel Dagobert"]=Duck
```

Elemente des Index können Leerzeichen sogar am Anfang oder Ende des Strings enthalten. Indexelemente, die nur aus Leerzeichen bestehen, sind jedoch nicht zulässig.

Ähnlich wie bei Perl oder PHP lassen sich auch mehrere Elemente in einem Rutsch definieren:

```
declare -A namen
namen=(
  [Donald]=Duck
  [Daniel]=Düsentrieb
  [Gustav]=Gans
  )
```

Auch beim Zugriff auf den Inhalt assoziativer Array verfährt man wie schon bei den indizierten Arrays. Dabei sei nochmals darauf hingewiesen, dass die Reihenfolge, in der die Elemente ich Array gespeichert werden, nicht durch die Reihenfolge der Eintragung festgelegt ist. Um an den Inhalt eines Hashes zu gelangen, wird der Key angegeben:

```
echo namen[Donald]
```

Anstelle eines konstanten Keys kann natürlich auch eine Shell-Variable verwendet werden, zum Beispiel:

```
01: Key=Gustav
02: echo ${namen[$Key]}
```

Ist die Variable »Key« nicht definiert, liefert die Bash die Fehlermeldung *»bad subscript«*.

**Sie müssen also darauf achten, dass alle Variablen, die Sie als Index verwenden, auch definiert sind!**

Ist dagegen der Index im Array nicht enthalten, liefert **`${namen[$Key]}`** einen leeren String. Um die Existenz eines Eintrags zu testen, können Sie folgendermaßen vorgehen:

```
01: if [ ${namen[$Key]+_} ]; then
02:   echo "$Key gefunden"
```

```
03: else
04:   echo "$Key nicht gefunden"
05: fi
```

In Zeile 1 wird die Parametersubstitution der Bash verwendet. Wenn das Hash-Element existiert, wertet die Bash den Ausdruck als »_« aus, andernfalls als Nullzeichenfolge. Wer will, kann auch »jawoll« oder irgend etwas anderes anstatt des Unterstrichs innerhalb der eckigen Klammern verwenden. Es gibt noch weitere Möglichkeiten in der Bash, auf Existenz oder Fehlen von Hashes zu testen, wobei sich alle auf die Parametersubstitution stützen. Das folgende Protokoll zeigt zwei weitere Alternativen zu der oben angegebenen:

```
$ ax[foo]="bar"
$ echo ${ax[foo]:-FEHLT}
"bar"
$ echo ${ax[nix]:-FEHLT}
FEHLT
$ echo ${ax[foo]:+EXISTIERT}
EXISTIERT
```

Alle Indexschlüssel (Keys) erhält man mittels **${!namen[@]}** und auf die Werte kann mit **${namen[@]}** zugegriffen werden. Der Befehl **echo ${!namen[@]}** ergibt dann für das obige Beispiel:

```
Daniel Gustav Donald
```

Die Werte dazu liefert der Befehl **${namen[@]}.** Man erhält die Ausgabe:

```
Düsentrieb Gans Duck
```

Die Länge eines assoziativen Arrays (im Prinzip ist dies die Anzahl der Schlüssel) erhält man mittels ${#MYMAP[@]}. Im Beispiel von oben erhält man also den Wert 3.

Am häufigsten wird man aber Schleifenkonstruktionen antreffen, die über Keys oder Werte iterieren. Die folgende for-Schleife läuft über die Keys:

```
01: for i in "${!namen[@]}"
02: do
03:   echo $i
04: done
```

Ohne das Ausrufezeichen vor dem Arraynamen wird über die Werte iteriert:

```
01: for w in "${namen[@]}"
02: do
03:   echo $w
04: done
```

Mit der folgenden Schleife werden schließlich alle Key-Werte-Paare ausgegeben:

```
01: for i in "${!namen[@]}"
02: do
03:   echo "$i --> ${namen[$i]}"
04: done
```

Startet man das Skript, erhält man:

```
Daniel --> Düsentrieb
Gustav --> Gans
Donald --> Duck
```

Die Reihenfolge entspricht nicht der Reihenfolge der Eintragung. Das kann auch mit einem weiteren Versuch demonstriert werden. Fügt man nun noch den Onkel Dagobert hinzu, liefert das Programm:

```
Onkel Dagobert --> Duck
Daniel --> Düsentrieb
Gustav --> Gans
Donald --> Duck
```

### *Demo-Adressverwaltung*

Nun ein etwas größeres Beispiel. Das folgende Programm stellt eine einfache Adressverwaltung dar, wobei die Namen den Schlüssel bilden und als Wert dann die Adresse im assoziativen Array eingetragen wird. Zur Verwaltung des Arrays stehen zwei einfache Funktionen zur Verfügung. Nach der Deklaration des Arrays »Adresse« als Hash in Zeile 1 wird in den Zeilen 3 bis 7 die Funktion `Speichere_Adresse` definiert, die Namen und Adresse als Parameter übernimmt und im Hash ablegt.

Die zweite Funktion, ***Suche_Adresse***, stellt zunächst fest, ob der Name als Schlüssel gespeichert ist, und gibt daraufhin die Adresse aus (Zeilen 10- 15). Kann der Name nicht gefunden werden, erfolgt eine Fehlermeldung und es wird der Rückgabewert auf 99 gesetzt (Zeilen 16 - 20).

In den Zeilen 22 bis 27 werden die Adressdaten aus der Datei **adressen.csv** eingelesen. Dort stehen pro Zeile Name und Adresse getrennt durch ein Semikolon. Diese dient dann auch nach dem Einlesen der Zeile als Trennzeichen für die beiden **cut-Befehle** (Zeilen 24 und 25), welche die beiden Felder an **Speichere_Adresse** übergeben.

In den Zeilen 29 - 36 werden dann die Adressdaten mittels **Suche_Adresse** abgefragt.

```
01: declare -A Adresse
02:
03: Speichere_Adresse ()
04:   {
05:   Adresse["$1"]="$2"
06:   return $?
07:   }
08:
09:
10: Suche_Adresse ()
11:   {
12:   if [ ${Adresse[$1]+_} ]
13:   then
14:     echo "$1's Adresse lautet ${Adresse[$1]}."
15:     return $?
16:   else
17:     echo "$1's Adresse ist nicht vorhanden."
18:     return 99
19:   fi
20:   }
21:
22: # Adressen aus CSV-Datei einlesen
23: while read LINE; do
24:   NAME=$(echo $LINE | cut -d';' -f1)
25:   ADDR=$(echo $LINE | cut -d';' -f2)
26:   Speichere_Adresse "$NAME" "$ADDR"
27: done < adressen.csv
28:
29: # Testaufrufe
30: Suche_Adresse "Donald Duck"
31: Suche_Adresse "Harry Potter"
32: Suche_Adresse "Jules Maigret"
33: Suche_Adresse "Sherlock Holmes"
34: Suche_Adresse "Herman Munster"
35: Suche_Adresse "Thomas A. Edison"
36: Suche_Adresse "John Doe"
```

***Das folgende Listing zeigt die Ausgabe des Programms.***

```
Donald Duck's Adresse lautet Flower Road 13, Duckburg, Callisota.
Harry Potter's Adresse lautet Privet Drive 4, Little Whinging, UK.
Jules Maigret's Adresse lautet Boulevard Richard-Lenoir 132, Paris, FR.
Sherlock Holmes's Adresse lautet Baker Street 221b, London, UK.
Herman Munster's Adresse lautet Mockingbird Lane 1313, Mockingbird Heights, USA.
Thomas A. Edison's Adresse lautet 37 Christie Street, Menlo Parc, NJ, USA.
John Doe's Adresse ist nicht vorhanden.
```

# Lösch- und Umwandlungsoperationen

Um ein assoziatives Array zu löschen, genügt es nicht, das Array neu zu deklarieren. Diese Re-Deklaration bewirkt eigentlich gar nichts, das folgende Programm gibt zweimal »bar« aus:

```
01: declare -A ARRAY
02: ARRAY[foo]=bar
03: echo ${ARRAY[foo]}
04:
05: declare -A ARRAY
06: echo ${ARRAY[foo]}
```

Sie müssen vielmehr das assoziative Array mittels `unset` gänzlich aus dem Gedächtnis der Bash entfernen und dann natürlich neu deklarieren. Beim Programm kommt so die Zeile 5 hinzu:

```
01: declare -A ARRAY
02: ARRAY[foo]=bar
03: echo ${ARRAY[foo]}
04:
05: unset ARRAY
06: declare -A ARRAY
07: echo ${ARRAY[foo]}
```

Meist will man jedoch nicht das ganze Array, sondern nur einzelne Einträge löschen. Dies erfolgt fast auf die gleiche Art und Weise wie oben, jedoch mit dem Unterschied, dass bei `unset` der Schlüssel des zu löschenden Eintrags angegeben wird. Beim folgenden Programm wird in Zeile 11 der Eintrag »zwei« gelöscht:

```
01: declare -A ARRAY
02: ARRAY=([eins]=one
03:        [zwei]=two
04:        [drei]=three)
05:
06: echo ${ARRAY[eins]}
07: echo ${ARRAY[zwei]}
08: echo ${ARRAY[drei]}
09: echo ""
10:
11: unset ARRAY[zwei]
12: echo ${ARRAY[eins]}
13: echo ${ARRAY[zwei]}
14: echo ${ARRAY[drei]}
```

Enthält der Schlüssel Leerzeichen oder andere Sonderzeichen, muss er in Gänsefüßchen oder Apostrophe eingeschlossen werden, was auch gilt, wenn der Schlüssel als Variable übergeben wird, z.B.:

```
unset ["Donald Duck"]  # Gaensefuesschen
unset ['Donald Duck']  # Apostroph
unset ["$Key"]         # Variable
```

Man kann sogar ein assoziatives Array in ein indiziertes Array konvertieren - wenn auch mit unvorhersagbarem Ergebnis, was die Reihenfolge betrifft. Im folgenden Beispiel wird ein assoziatives Array deklariert, gefüllt und ausgegeben (Zeilen 1 bis 9). In Zeile 11 erzeugt die Zuweisung aus ARRAY ein indiziertes Array KEYS. Beachten Sie bitte die runden Klammern, die den Effekt bewirken.

```
01: declare -A ARRAY
02: ARRAY=(["eins"]="one"
03:        ["zwei"]="two"
04:        ["drei"]="three")
05:
06: echo "Ausgabe ARRAY"
07: for K in ${!ARRAY[@]}; do
08:   echo "$K --> ${ARRAY[$K]}"
09: done
10:
11: KEYS=( ${!ARRAY[@]} )
12:
13: echo ""
14: echo "Ausgabe KEYS"
15: for K in 0 1 2; do
16:   echo "$K --> ${KEYS[$K]}"
17: done
```

Die Ausgabe des indizierten Arrays in den Zeilen 14 bis 17 zeigt, dass die Reihenfolge der Speicherung im Hash wieder mal nicht der Eingabereihenfolge entsprach und daher das indizierte Array vielleicht nicht ganz den Vorstellungen des Programmierers entspricht.

```
Ausgabe ARRAY
zwei --> two
eins --> one
drei --> three

Ausgabe KEYS
0 --> zwei
1 --> eins
2 --> drei
```

## Anwendungen für Hashes

Wofür sind die assoziativen Arrays (Hashes) eigentlich wirklich gut? Hashes sind nützlicher als Arrays, wenn man auf ein Element lieber mit einer expliziten Bezeichnung (einem Schlüssel) als mit einem bloßen numerischen Index zugreifen möchte. Eine Anwendung ist das Prüfen, ob Schlüssel(-Worte) beispielsweise in einer Datei vorhanden sind. In solche Fällen kommt es nur auf den Schlüssel und nicht auf den Wert an. Ein typisches Beispiel wäre das Erstellen einer Wortliste aus einer Datei. Hierbei kommt es nur auf die Worte an und nicht auf deren Auftretenshäufigkeit. das folgende Programm hat eine ähnliche Funktion wie das UNIX-Kommando `uniq`. Es wird jedes Wort nur einmal gespeichert, egal wie häufig es vorkommt. Bei dem Beispielprogramm wird davon ausgegangen, dass in einer Datei jeweils nur ein Wort in jeder Zeile steht.

Die ganze Magie verbirgt sich in Zeile 5. Das gelesene Wort wird als Schüssel gespeichert, wobei der Wert beliebig sein darf. Ist der Schlüssel neu, bekommt der Hash ein weiteres Element, ist er dagegen schon vorhanden, wird der Hash nicht erweitert. Die Ausgabe (Zeilen 8 bis 11) kennen Sie ja schon aus anderen Programmen. Diesmal wurde nur eine Sortierung nachgeschaltet.

```
01: declare -A STAT
02:
03: # Worte einlesen
```

```
04: while read WORT; do
05:   STAT[$WORT]=1
06: done < text
07:
08: # Kontrollausgabe
09: for I in ${!STAT[@]}; do
10:   echo "$I"
11: done | sort
```

Es gehört zwar nicht direkt zu Thema, aber vielleicht fragt sich manch einer, wie man eine normale Textdatei in Worte so aufsplittet, dass sich in jeder Zeile des Ergebnisses genau ein Wort befindet. Hier hilft das Tool `tr`. Das folgende Skript (eigentlich ein Einzeiler, der nur zur Erläuterung hier mehrzeilig geschrieben wird) erstellt aus einer beliebigen Textdatei eine Wortliste. In Zeile 1 werden mehrere aufeinander folgende Leerzeichen durch ein einziges ersetzt. Zeile 2 entfernt alle Satzzeichen und Zahlen (alles bis auf die Buchstaben), in Zeile 3 werden die Leerzeichen zu Newline-Zeichen. Schließlich wird noch sortiert und dann die Doubletten entfernt.

```
01: cat $DATEI | tr -s ' ' ' ' |
02:  tr -d ' '-'@' |
03: tr ' ' '\n' |
04: sort | uniq
```

Eine weitere typische Anwendung ist das Erstellen einer Häufigkeits-Statistik. Bei jedem Auftreten eines Schlüssels wird diesmal der Wert inkrementiert und schon hat man die Häufigkeiten der Schlüssel ermittelt. Das soll anhand einer fiktiven Notenstatistik demonstriert werden. Fiktiv ist die Statistik deshalb, weil ich mir die Datendatei auch per Skript und der `RANDOM`-Variablen erzeugt habe. Deshalb gibt es auch keine Normalverteilung.

Zuerst werden die Noten wieder per Schleife in den Zeilen 8 bis 11 eingelesen. Im Gegensatz zum Beispiel oben wird jedoch der Wert inkrementiert (Zeile 9). Deshalb listet die Kontrollausgabe (Zeilen 13 bis 19) diesmal auch Schlüssel und Wert auf. Womit der gewünschte Zweck eigentlich schon erfüllt wäre.

Weil aber bei der Ausgabe von Häufigkeiten gerne ein Histogramm erzeugt wird, habe ich das Skript entsprechend aufgepimpt. Zuerst wird (Zeilen 21 bis 27) das Maximum der Werte ermittelt. Dann kann nämlich die Länge der Histogramm-Balken normiert werden. Die Variable `HMAX` legt die maximale Länge fest, sie ist in Zeile 4 definiert. Zusammen mit dem ermittelten Maximum `MAX` kann dann die Länge des Balkens für einen bestimmten Wert nach der Formel

```
LEN = (Wert * HMAX) / MAX
```

ermittelt werden. Das wird in der Schleife über alle Schlüssel (Zeilen 29 bis 38) erledigt. Um das Histogramm zu »zeichnen« wird in der inneren Schleife (Zeilen 34 bis 36) `LEN` mal ein Doppelkreuz ausgegeben. Ganz zum Schluss wird das Histrogramm nach Noten sortiert.

```
01: declare -A STAT
02:
03:  # Max. Laenge eines Histogrammbalkens
04:  HMAX=40
05:
06: # Noten einlesen
07: COUNT=0
08: while read NOTE; do
09:   STAT[$NOTE]=$(( ${STAT[$NOTE]} + 1 ))
10:   COUNT=$(( $COUNT + 1 ))
11: done < noten
```

```
12:
13: # Ausgabe: Note, Anzahl (unsortiert)
14: for I in ${!STAT[@]}; do
15:   echo "$I ${STAT[$I]}"
16: done
17:
18: # Gesamtanzahl ausgeben
19: echo $COUNT
20:
21: # Maximalwert der Anzahlen bestimmen
22: MAX=0
23: for I in ${!STAT[@]}; do
24:   if [ ${STAT[$I]} -gt $MAX ]; then
25:     MAX=${STAT[$I]}
26:   fi
27: done
28:
29: # Histogramm ausgeben (sortiert)
30: echo ""
31: for I in ${!STAT[@]}; do
32:   echo -n "$I (${STAT[$I]}): "
33:   LEN=$(( ${STAT[$I]} * $HMAX / $MAX ))
34:   for K in $(seq 1 $LEN); do
35:     echo -n "#"
36:   done
37:   echo ""
38: done | sort
```

Zum Testen habe ich mal 1000 Notenwerte generiert, die klaglos vom Skript geschluckt werden. Die Ausgabe zeigt, wie schon erwähnt, nicht die zu erwartende Normalverteilung, sondern eine relativ ordentliche Gleichverteilung.

```
3,7 81
5,0 82
3,3 76
3,0 86
2,7 87
2,3 76
2,0 81
4,0 70
4,3 65
4,7 61
1,7 71
1,0 81
1,3 83
1000
```

```
1,0 (81): ####################################
1,3 (83): #####################################
1,7 (71): ##############################
2,0 (81): ####################################
2,3 (76): ##############################
2,7 (87): #######################################
3,0 (86): ####################################
3,3 (76): ##############################
3,7 (81): ####################################
4,0 (70): ##############################
4,3 (65): ############################
4,7 (61): ############################
5,0 (82): ####################################
```

Eine Bemerkung zum Schluss: Wenn Sie anfangen, mit der Bash zu spielen und gerade verzweifeln, weil Ihr Skript nicht läuft, denken Sie an den Aufruf `bash -vx skriptname`. Hier listet die Bash auf, wie bei jeder Zeile die Parameter ersetzt werden und was bei der Ausführung dieser Zeile geschieht. Wenn das immer noch

nicht hilft, machen Sie es wie früher die BASIC-Programmierer. Zeile für Zeile eingeben und ständig das Programm laufen lassen (programming by trial and error).

## Referenzen

Carl Albing, J.P. Vossen, Cameron Newham: bash Cookbook - Solutions and Examples for bash Users, O'Reilly, 2007

## Autoreninformation

Jürgen Plate ist Professor für Elektro- und Informationstechnik an der *Hochschule für angewandte Wissenschaften München*. Er beschäftigt sich seit 1980 mit Datenfernübertragung und war, bevor der Internetanschluss für Privatpersonen möglich wurde, in der Mailboxszene aktiv. Unter anderem hat er eine der ersten öffentlichen Mailboxen - TEDAS der *mc*-Redaktion - programmiert und 1984 in Betrieb genommen.

Dieser Artikel ist zuerst erschienen in UpTimes, Mitgliederzeitschrift des GUUG e.V., Ausgabe 2017-1. Veröffentlichung mit freundlicher Genehmigung.

-

Referenzen:

- Advanced Bash-Scripting Guide
- Bash Style Guide und Kodierungsrichtlinie
- Bash Hackers Wiki

https://codefather.tech/blog/

# *Übung zu „shift“- und „array“-Funktion in der bash*

## *1. Aufgabe*

Welche Funktion zeigt nachfolgendes Skript?
Testen Sie die Funktion des bash-Skriptes und erläutern Sie die darin enthaltenen Befehle! - Wie viele Set-Parameter können einem Skript mitgegeben werden können?

```
1 #! /bin/bash
2 if   [[   $#   -lt  11   ]]
3 then
4   echo "Du Döddele solltest doch mindestens 11 Parameter übergeben"
5 else
6 z=$#   # Es wird die Variable z mit dem Inhalt von $# gefüllt z.B. 15
7 while  (( 0 < $# ))
8     do
9 #    echo "\$1=$1"
10          felder[$#]=$1   # felder[15]=1
11       #   nach dem ersten Durchlauf
12       #   felder[15]=1   Bei Übergabe von: 1 2 .... 8 9 A B .. E F
13       #   zweiter Durchlauf
14       #   felder[14]=2   als Set-Parameter
15       #   ...
16       #   felder[1]=F
17          shift
18       done
19       echo -e "rueckwaerts: \c"
20       i=0
21       while  (( i < z ))
22         do
23           ((i=i+1))
24       ###    echo "feld[$i] = ${felder[$i]}"
25           echo -e "${felder[$i]} \c"
26         done
27         echo " "
28       fi
```

***2. Aufgabe***.

Hier eine weitere Übung!

```
! /bin/bash
clear
    1 echo "#-----------------------------------------"
    2 echo "#    Erläutere und erkläre das Beispiel    "
    3 echo "#    hier so ausführlich wie nur möglich!     "
    4 echo "#    Verfasse hierzu #Anmerkungen im Skript!  "
    5 echo "#    Welche Übergabewerte als PArameter er-   "
    6 echo "#    scheinen Ihnen sinnvoll?                 "
    7 echo "#-----------------------------------------"
    8 echo "# \$1=\"Hallo Egon Egon\"    "
    9 echo "starte das Skript mal z. B. mit: arrayuebung01 HalloHallo Egon
      Egon"
   10       if [[  $#  == 0 ]]
   11       then
   12         set "Hallo Hallo Egon Egon"
   13       fi
   14       Name="$1"
   15       Begruessung="${Name%Egon}"
   16       echo "Hier folgt die Begruessung $Begruessung"
   17       echo "Der Befehl: echo \"\$1\" | sed \"s/Egon//\" hat "
   18       echo "          die geliche Funktion"
   19       echo "\${Name%%Egon} ergibt: ${Name%%Egon}"
   20       echo "\$Name#Hallo} ergibt: ${Name#Hallo}"
   21       echo "\${Name##Hallo} ergibt: ${Name##Hallo}"
   22       echo "\${1:-15} ergibt:  ${1:-15}"
   23       echo "Bedenke \$7 und \$8 wurde nicht gestzt!"
   24       echo "\$7 ist jetzt:  ${7:-15} und nicht ${8:--9}"
   25       echo "Fritz"  |  tr  [:upper:]   [:lower:]
```

## *3. Aufgabe*

Beschreiben Sie in Stichpunkten (befehlszeilenbezogen -> Nr) welche Bedeutung nachfolgender Code generiert!
Betrachten Sie die Zeile-Nr. 13 bzw. 20 bis Zeile-Nr. 19 bzw. 25 im Funktionszusammenhang!

```
1       #! /bin/bash
2       echo "das ist der \$1-Set-Parameter: ${1:-15}"
3       felder=("a1" "a2" "a3" "01234")
4       echo ${felder[2]}
5       echo "Das Array \$felder hat ${#felder[*]} Felder"
6       echo "Der Inhalt von Array-Feld Nr 4 ist ${#felder[3]} Zeichen lang"
7       echo "Der Arrayfeldinhalt Nr 3   hat eine Länge von"
8       echo "${#felder[2]} Zeichen"
9       felder[4]="hans"
10      felder[5]="otto"
11      felder[6]="anton"
12      i=0
13      while  (( i < "7" ))
14        do
15          echo ${felder[$i]}
16          ((i=i+1))
17        done
18      echo "noch einmal"
19      i=0
20      while  (( i < ${#felder[*]} ))
21        do
22          echo ${felder[$i]}
23          ((i=i+1))
24        done
```

Lösung:

1.

2.

3.

4.

5.

6.

7.

8.

9.

Die generierte Ausgabe auf der Bash-Konsole:

```
das ist der $1-Set-Parameter: 15
a3
Das Array $felder hat 4 Felder
Der Inhalt von Array-Feld Nr 4 ist 5 Zeichen lang
Der Arrayfeldinhalt Nr 3   hat eine Länge von
2 Zeichen
a1
a2
a3
01234
hans
otto
anton
noch einmal
a1
a2
a3
01234
hans
otto
anton
```

```
                    1
                  1   2
                1   2   3
              1   2   3   4
            1   2   3   4   5
```

```
echo $felderZeile5
```

### *4. Aufgabe*

Binden Sie die nachfolgenden Fragmente in sinnvole Codezeilen ein und erläutern sie deren Funktion(en)!

${Variable%Muster}  →

${Variable%%Muster}  →

${Variable#Muster}  →

${Variable##Muster}  →

${1:-15}  →

echo “Fritz“ | tr [:upper:] [:lower:]