ADMIN:DOCS

Bash Grundkurs #2: Verwendung Von Variablen In Bash

Lerne den professionellen Umgang mit Bash-Variablen: Deklaration, Zuweisung, String-Quoting, Parameter-Expansion, Arrays, Umgebungsvariablen und Prozessvererbung.

Kategorie: Bash-Grundlagen **Veröffentlicht:** 23.10.2024 **Zuletzt geprüft:** 10.11.2024

Willkommen zum zweiten Teil unserer technischen Serie über Linux und Bash-Programmierung!

Nachdem wir in Bash Grundkurs #1: Erstelle Dein Erstes Bash-Shell-Skript die Shebang-Mechanik, Ausführungsberechtigungen und den Bash Strict Mode kennengelernt haben, widmen wir uns heute dem Herzstück dynamischer Skripte: **der Speicherung und Verarbeitung von Variablen**.

Statische Skripte, die stets dieselben hartkodierten Pfade ausführen, stoßen im Administrationsalltag schnell an ihre Grenzen. Erst durch Variablen werden deine Skripte flexibel, modular und wiederverwendbar. Sie nehmen Benutzereingaben entgegen, speichern Pfade, zählen Durchläufe, fangen Befehlsausgaben ab und verarbeiten dynamische Datenstrukturen.

In dieser Lektion lernst du alles über die richtige Deklaration, sauberes Quoting zur Vermeidung von Sicherheitslücken, Funktions-Scopes (`local`), Umgebungsvariablen (`export`), spezielle Positions-Parameter sowie mächtige **Parameter-Expansionen**, mit denen du Strings ohne langsame Hilfswerkzeuge wie `sed` oder `awk` manipulieren kannst.

> **Hinweis zum Bash-Typensystem:** Im Gegensatz zu Sprachen wie C, Rust oder Java ist die Bash standardmäßig **schwach typisiert**. Jede Variable wird intern zunächst als

Quelle: https://admindocs.de/de/bash-grundlagen/bash-grundkurs-verwendung-von-variablen-in-bash

> reiner Text-String gespeichert. Erst bei bestimmten Rechenoperationen oder Vergleichen interpretiert die Shell den Inhalt kontextabhängig als Ganzzahl.

Deklaration & Zuweisung: Die Grundlagen

In der Bash deklarierst du eine Variable einfach durch Angabe ihres Namens, gefolgt von einem Gleichheitszeichen (`=`) und dem zuzuweisenden Wert:

```
# Zeichenketten (Strings)
benutzername="admin"
server_ip="192.168.1.50"

# Ganzzahlen (Integer)
max_versuche=5
port=8080
```

> ⚠ **Wichtige Syntaxregel:** Zwischen dem Variablennamen, dem Gleichheitszeichen und dem Wert dürfen **niemals Leerzeichen** stehen!

```
name = "Max"   # FEHLER: Bash versucht den Befehl 'name' mit Argument '=' auszuführen
name= "Max"    # FEHLER: Setzt leere Variable 'name' und führt Befehl 'Max' aus
name="Max"     # KORREKT!
```

Namenskonventionen in der Shell

- **Großbuchstaben (`BACKUP_DIR`, `PORT`, `TIMEOUT`):** Werden konventionsgemäß für globale Konstanten, Systemvariablen und exportierte Umgebungsvariablen genutzt.
- **Kleinbuchstaben (`dateiname`, `index`, `schleifen_zaehler`):** Werden für interne, temporäre und lokale Skriptvariablen verwendet.
- **Erlaubte Zeichen:** Buchstaben (`a-z`, `A-Z`), Ziffern (`0-9`) und der Unterstrich (`_`). Der Name darf **nicht mit einer Ziffer** beginnen.

Auf Variablen zugreifen: `$VAR` vs. `${VAR}`

Um den Inhalt einer Variable auszulesen, stellst du dem Namen ein Dollarzeichen (`$`) voran:

```
echo $benutzername
echo "Verbinde mit $server_ip auf Port $port..."
```

Für sauberen und eindeutigen Code solltest du dir angewöhnen, Variablennamen in **geschweifte Klammern** zu setzen (`${VAR}`):

```
log_name="backup"
# Eindeutige Trennung vom nachfolgenden Text:
echo "${log_name}_2024.tar.gz"  # Ausgabe: backup_2024.tar.gz

# Ohne Klammern sucht die Shell nach der nicht existierenden Variable $log_name_2024:
echo "$log_name_2024.tar.gz"    # FEHLER: Gibt nur '.tar.gz' aus!
```

Quelle: https://admindocs.de/de/bash-grundlagen/bash-grundkurs-verwendung-von-variablen-in-bash

Quoting-Regeln: Single Quotes vs. Double Quotes

Das korrekte Quoting ist der wichtigste Sicherheitsaspekt beim Umgang mit Variablen in der Bash:



Praxisvergleich:

```
projekt="Admin Docs"

# Double Quotes: Variable wird aufgelöst
echo "Projekt: $projekt"
# Ausgabe: Projekt: Admin Docs

# Single Quotes: Text bleibt exakt wie getippt
```

```
echo 'Projekt: $projekt'
# Ausgabe: Projekt: $projekt

# Die Tücke fehlender Quotes bei Dateinamen mit Leerzeichen:
dateiname="mein dokument.pdf"
rm $dateiname    # FEHLER: Versucht 'mein' und 'dokument.pdf' getrennt zu löschen!
rm "$dateiname"  # KORREKT: Behandelt den Inhalt als genau ein Datei-Argument
```

> **Goldene Regel:** Schließe Variablen beim Zugriff im Fließtext und bei Befehlsaufrufen **grundsätzlich in doppelte Anführungszeichen** ein (`"$VAR"` oder `"${VAR}"`).

Gültigkeitsbereiche: Lokale vs. Globale Variablen

Standardmäßig sind alle in einem Bash-Skript definierten Variablen **global**. Das bedeutet: Wurde eine Variable deklariert, ist sie im gesamten restlichen Skript les- und überschreibbar – selbst wenn sie innerhalb einer Funktion zugewiesen wurde.

```
┌──────────────────────────────────────────────────────┐
│                VARIABLE SCOPING IN FUNKTIONEN        │
├──────────────────────────────────────────────────────┤
│                                                      │
│  [ Hauptprogramm / Skript-Ebene ]                    │
│  GLOBAL_VAR="aktiv"                                  │
│         │                                            │
│         ├── Funktionsaufruf: worker()                │
│         │     ├── local TEMP_VAR="lokal" (lokal gültig)
│         │     └── GLOBAL_VAR="überschrieben" (GEFAHR!)
│         │                                            │
│         ▼                                            │
│  Nach dem Funktionsaufruf:                           │
```

```
│  ├── TEMP_VAR ist wieder WEG (leer)                          │
│  └── GLOBAL_VAR wurde global verändert                       │
│                                                              │
└──────────────────────────────────────────────────────────────┘
```

Saubere Kapselung mit `local` in Funktionen

Um ungewollte Seiteneffekte in größeren Administrationsskripten zu verhindern, solltest du Variablen innerhalb von Funktionen immer mit dem Schlüsselwort `local` deklarieren:

```
#!/usr/bin/env bash
set -euo pipefail

system_status="ONLINE"

pruefe_speicher() {
    # Lokale Variable - existiert NUR innerhalb dieser Funktion:
    local freier_speicher
    freier_speicher=$(df -h / | awk 'NR==2 {print $4}')

    echo "Freier Speicher: $freier_speicher"
}

pruefe_speicher

# Versuch, von außen auf die lokale Variable zuzugreifen:
# Dank 'set -u' bricht das Skript hier mit einem Fehler ab, da $freier_speicher lokal ist!
```

Umgebungsvariablen & Prozessvererbung (`export`)

Normale Shell-Variablen existieren nur im Speicher der aktuellen Skript-Instanz. Wird aus deinem Skript heraus ein Unterprogramm oder ein anderes Skript gestartet, kennt der Kindprozess diese Variablen nicht.

Quelle: https://admindocs.de/de/bash-grundlagen/bash-grundkurs-verwendung-von-variablen-in-bash

Mit dem Befehl `export` beförderst du eine Variable in die **Umgebung** (*Environment*):

```
# Variable definieren und exportieren
export DATABASE_URL="mysql://db.local:3306/production"
export APP_ENV="production"
```



> ⚠ **Wichtige Einbahnstraße:** Ein Kindprozess erbt zwar die exportierten Variablen seiner Eltern-Shell, kann diese jedoch **niemals in der Eltern-Shell verändern**. Änderungen im Kindprozess bleiben strikt auf diesen isoliert.

Spezielle automatische Bash-Variablen

Die Bash stellt eine Reihe vordefinierter, schreibgeschützter Variablen bereit, die wichtige Informationen über den aktuellen Skriptlauf enthalten:

Variable	Bedeutung & Anwendungsfall
`$0`	Name oder Pfad des aktuell ausgeführten Skripts
`$1`, `$2` .. `$9`	Die übergebenen Positions-Parameter / Argumente
`${10}`, `${11}`	Argumente ab Position 10 (müssen in geschweiften Klammern stehen!)
`$#`	Anzahl aller an das Skript übergebenen Argumente
`"$@"`	Alle übergebenen Argumente als **separate, gequotete Wörter**
`"$*"`	Alle Argumente als **ein einziger zusammenhängender String**
`$$`	Prozess-ID (PID) des aktuell laufenden Skripts
`$!`	Prozess-ID des zuletzt im Hintergrund gestarteten Prozesses
`$?`	Exit-Status / Rückgabecode des unmittelbar zuvor ausgeführten Befehls

Der entscheidende Unterschied: `"$@"` vs. `"$*"`

Die Unterscheidung zwischen `"$@"` und `"$*"` ist eine der häufigsten Fallstricke beim Weiterleiten von Parametern an andere Befehle:

```
#!/usr/bin/env bash
# Skriptaufruf: ./test.sh "Mein Dokument.pdf" "Bild 1.png"

# 1. Verwendung von "$@": Erhält jedes Argument einzeln gequotet:
for datei in "$@"; do
    echo "Verarbeite Datei: $datei"
done
```

```
# Durchlauf 1: Mein Dokument.pdf
# Durchlauf 2: Bild 1.png

# 2. Verwendung von "$*": Fügt alle Argumente zu EINEM String zusammen:
for datei in "$*"; do
    echo "Verarbeite Datei: $datei"
done
# Durchlauf 1: Mein Dokument.pdf Bild 1.png (FALSCH!)
```

> **Best Practice:** Nutze für das Weiterleiten oder Iterieren von Skript-Argumenten immer `"$@"`.

Parameter-Expansion: Strings manipulieren ohne Hilfswerkzeuge

Die Bash verfügt über eine extrem leistungsfähige interne String-Manipulation namens **Parameter-Expansion**. Damit sparst du dir langsame Subprozess-Aufrufe wie `sed`, `awk` oder `cut`.

1. Fallback-Werte & Fehlerprüfung

```
# 1. Fallback-Wert nutzen, wenn Variable leer oder ungesetzt ist (${VAR:-default}):
BENUTZER="${1:-gast}"
echo "Angemeldet als: $BENUTZER"

# 2. Variable dauerhaft mit Standardwert belegen, falls ungesetzt (${VAR:=default}):
echo "Speicherort: ${LOG_DIR:=/var/log/app}"

# 3. Skript mit Fehlermeldung abbrechen, wenn Variable fehlt (${VAR:?fehler}):
# Verhindert fatale Unfälle wie rm -rf /* !
```

```
TARGET_DIR="${1:?Fehler: Es muss ein Zielverzeichnis angegeben werden!}"
```

2. String-Länge und Slicing (Teilstrings)

```
TEXT="Linux-Administration"

# Länge des Strings ermitteln (${#VAR})
echo "Zeichenanzahl: ${#TEXT}"  # Ausgabe: 20

# Teilstring ab Position extrahieren (${VAR:offset:length})
echo "${TEXT:0:5}"   # Ausgabe: Linux
echo "${TEXT:6}"     # Ausgabe: Administration (ab Index 6 bis Ende)
```

3. Suchen & Ersetzen

```
PFAD="/var/www/html/app/config.php"

# Erstes Vorkommen ersetzen (${VAR/such/ersatz}):
echo "${PFAD/html/public}"     # /var/www/public/app/config.php

# Alle Vorkommen ersetzen (${VAR//such/ersatz}):
IP="192.168.1.1"
echo "${IP//./-}"              # 192-168-1-1
```

4. Präfix- und Suffix-Stripping (Dateiendungen & Pfade trennen)

Dies ist eines der unverzichtbarsten Werkzeuge für Automatisierungsskripte:

```
DATEIPFAD="/var/backups/webserver_database.tar.gz"

# Dateinamen ohne Verzeichnispfad extrahieren (Größtes Präfix bis / entfernen: ##*/)
DATEINAME="${DATEIPFAD##*/}"
echo "$DATEINAME"  # Ausgabe: webserver_database.tar.gz

# Verzeichnispfad extrahieren (Größtes Suffix ab / entfernen: %/*)
VERZEICHNIS="${DATEIPFAD%/*}"
echo "$VERZEICHNIS"  # Ausgabe: /var/backups

# Dateiendung entfernen (Kürzestes Suffix .gz entfernen: %.gz)
echo "${DATEINAME%.gz}"  # Ausgabe: webserver_database.tar
```

5. Groß- und Kleinschreibung konvertieren (Bash 4+)

```
MODUS="production"
echo "${MODUS^^}"  # Ausgabe: PRODUCTION (Alles in Großbuchstaben)

TITEL="DATABASE"
echo "${TITEL,,}"  # Ausgabe: database (Alles in Kleinbuchstaben)
```

Arrays in der Bash: Listen & Key-Value-Speicher

Ab Bash Version 4 stehen dir vollwertige indizierte und assoziative Arrays zur Verfügung:

1. Indizierte Arrays (Nummerierte Listen)

```
# Array deklarieren und befüllen
SERVER_LISTE=("web01" "web02" "db01" "cache01")

# Erstes Element abrufen (0-indiziert):
echo "Erster Server: ${SERVER_LISTE[0]}"

# Alle Elemente ausgeben (${ARRAY[@]}):
echo "Alle Server: ${SERVER_LISTE[@]}"

# Anzahl der Elemente ermitteln (${#ARRAY[@]}):
echo "Anzahl Server: ${#SERVER_LISTE[@]}"

# Neues Element anhängen:
SERVER_LISTE+=("mail01")

# Über das Array iterieren:
for server in "${SERVER_LISTE[@]}"; do
    echo "Pinge Server: $server..."
done
```

2. Assoziative Arrays (Key-Value-Hashes)

Assoziative Arrays müssen vor ihrer Verwendung explizit mit `declare -A` deklariert werden:

```
# Assoziatives Array deklarieren
declare -A SERVER_PORTS

# Werte zuweisen
SERVER_PORTS["http"]=80
SERVER_PORTS["https"]=443
```

```
SERVER_PORTS["ssh"]=22

# Wert über Schlüssel abrufen
echo "HTTPS-Port ist: ${SERVER_PORTS["https"]}"

# Alle Schlüssel (Keys) auslesen (${!ARRAY[@]}):
for dienst in "${!SERVER_PORTS[@]}"; do
    echo "Dienst $dienst läuft auf Port ${SERVER_PORTS[$dienst]}"
done
```

Konstanten & typisierte Variablen mit `readonly` und `declare`

Wenn Variablen vor versehentlichem Überschreiben geschützt oder als reine Zahlen typisiert werden sollen, nutzt du `readonly` oder `declare`:

```
# Unveränderbare Konstante deklarieren
readonly BACKUP_DIR="/mnt/secure_backups"
# Jeder spätere Zuweisungsversuch (BACKUP_DIR="/tmp") führt zum Skriptabbruch!

# Variable als Ganzzahl (Integer) typisieren (declare -i)
declare -i ZAEHLER=10
ZAEHLER+=5  # Führt automatisch eine mathematische Addition aus (15), keine String-Verkettung!
echo "Zähler: $ZAEHLER"
```

Übungen & Praxis-Check

> **Praxisaufgaben zur Lektion #2:**
>
> 1. **Aufgabe 1 (Pfad- und Namens-Parser):**

> Schreibe ein Skript `parse_file.sh`, das einen vollständigen Dateipfad (z. B. `/var/log/nginx/access.log.gz`) als `$1` entgegennimmt und per Parameter-Expansion das Verzeichnis, den reinen Dateinamen sowie die Dateiendung extrahiert und ausgibt.
>
> 1. **Aufgabe 2 (Array-Server-Check):**
>
> Erstelle ein Skript, das ein Array von 3 Domainnamen definiert (`heise.de`, `github.com`, `admindocs.de`), über diese iteriert und mit `ping -c 1` prüft, ob der Host erreichbar ist.
>
> 1. **Aufgabe 3 (Sicheres Backup-Verzeichnis):**
>
> Schreibe ein Skript, das das Zielverzeichnis mit `${1:?Fehler: Kein Zielpfad übergeben!}` erzwingt und einen Fallback-Benutzernamen mit `${2:-admin}` setzt.

Musterlösung zu Aufgabe 1:

```
#!/usr/bin/env bash
set -euo pipefail
IFS=$'\n\t'

EINGABE_PFAD="${1:?Bitte einen Dateipfad als Parameter übergeben!}"

VERZEICHNIS="${EINGABE_PFAD%/*}"
VOLLER_NAME="${EINGABE_PFAD##*/}"
DATEI_ENDUNG="${VOLLER_NAME##*.}"
BASENAME="${VOLLER_NAME%.*}"

echo "=== DATEI-ANALYSE ==="
echo "Eingabepfad : $EINGABE_PFAD"
echo "Verzeichnis : $VERZEICHNIS"
echo "Dateiname   : $VOLLER_NAME"
echo "Basisname   : $BASENAME"
echo "Endung      : $DATEI_ENDUNG"
echo "====================="
```

```
exit 0
```

Befehlsreferenz (Cheatsheet)

Syntax / Befehl	Funktion & Auswirkung
`VAR="Wert"`	Weist der Variable einen Wert zu (keine Leerzeichen!)
`"${VAR}"`	Liest den Inhalt der Variable sicher aus
`local VAR="x"`	Beschränkt die Gültigkeit der Variable auf die lokale Funktion
`export VAR="x"`	Macht die Variable für gestartete Kindprozesse verfügbar
`readonly VAR="x"`	Markiert die Variable als unveränderbare Konstante
`declare -A HASH`	Deklariert ein assoziatives Array (Key-Value)
`"${VAR:-standard}"`	Verwendet Standardwert, falls `VAR` leer oder ungesetzt ist
`"${VAR:?fehler}"`	Bricht mit Fehlermeldung ab, falls `VAR` leer oder ungesetzt ist
`"${#VAR}"`	Gibt die Anzahl der Zeichen des Strings zurück
`"${VAR##*/}"`	Entfernt den gesamten Pfad bis zum letzten Slash (Dateiname)
`"${VAR%/*}"`	Entfernt den Dateinamen ab dem letzten Slash (Verzeichnispfad)
`"${VAR^^}"`	Wandelt alle Zeichen in Großbuchstaben um
`"${VAR,,}"`	Wandelt alle Zeichen in Kleinbuchstaben um
`"$@"`	Alle Skriptargumente als separat geschützte Parameter
`"$#"`	Anzahl der an das Skript übergebenen Parameter

Weiterführende Ressourcen

Ressource	Beschreibung
GNU Bash Parameter Expansion	Offizielle GNU-Referenz aller String- und Expansionsoperatoren
Bash-Grundkurs #1: Erstes Skript	Der erste Teil unseres Grundkurses: Shebang, Rechte und Strict Mode
Linux Befehlszeilenprozessor Leitfaden	Fundamentales Wissen zu Shells, I/O-Streams und Pipes
chmod & Dateiberechtigungen	Linux-Berechtigungskonzepte im Detail

Fazit

Variablen und Parameter-Expansionen sind das Fundament, um Shell-Skripte von simplen Befehlsketten in intelligente, flexible Administrationswerkzeuge zu verwandeln. Wer String-Manipulationen nativ in der Bash über `${VAR##*/}` oder `${VAR:-default}` löst, schreibt nicht nur robustere Skripte, sondern profitiert auch von spürbar schnelleren Ausführungszeiten.

> **Tipp:** Schließe Variablenzugriffe immer in Anführungszeichen ein (`"${VAR}"`) und nutze `${VAR:?}` bei kritischen Operationen wie `rm -rf` . Verwende innerhalb von Funktionen konsequent `local` , um unbemerkte globale Variablenüberschreibungen auszuschließen.

In der nächsten Lektion bringen wir Entscheidungslogik in unsere Skripte und lernen, wie Verzweigungen und Schleifen gesteuert werden: **Weiter geht es mit:** Bash Grundkurs #3: Kontrollstrukturen In Bash

Zur Kursübersicht: Alle Lektionen des Bash-Grundkurses