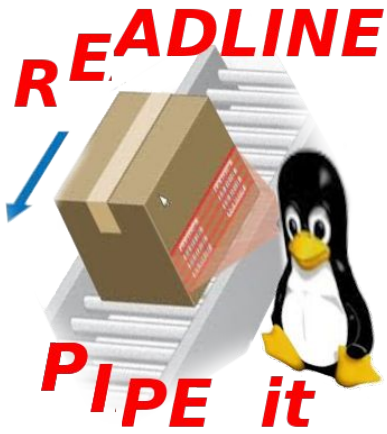
ein-ausblenden

Wir lesen Eingabezeilen, Namen, Dateien, Daten usw. mit der „bash“-Shell



1. Aufgabe

Lesen Sie eine CSV-Datei, die lauter Benutzer enthält ein und generieren Sie Damit Benutzer auf dem linux-System.

Was geschieht hier in Skript areadline2?

```
#!/bin/bash

# Die Datei, die als Argument $1 übergeben wurde,
#                                    soll zeilenweise eingelesen werden
cat $1 | while read variable
do
  echo $variable
done
```

Text 1: Skript: areadline2: Kommando-Ausgabe | while read line → do

Die übergebene Datei Namensliste sieht z. B. wie folgt aus:

```
Meier,Peter,03.05.1999,Hofgarten,33,91555,Ansbach,09123-88842
Huber,Rudi,03.06.2000,Hufgasse,3,91122,Schwabach,09122-82441
Schmid,Hans,02.07.1998,Zu den Gründen,1,95355,Röttenbach,0911-842
Kerner,Otto,31.10.1997,Am Bahnhof,1,91421,Hallstadt,09993-83122
```

Text 2: Dateiinhalt der CSV-Datei: „Adressliste.csv“

Führen Sie nachfolgenden Test durch!
Die letzte Zeile der csv-Adressliste ist nicht mit einem Return abgeschlossen. - Was beobachten Sie hinsichtlich des Einlesevorganges bei der Anwendung obigen Skriptes? - Wozu dient der Befehl: ***„mapfile“?***

2. Aufgabe

Erkläre die inline-Eingabe-Umleitung bzw. das so genannte Here-Dokument!

Geben Sie hierzu einfach nachfolgende Zeilen in ein Terminal mit gestarteter bash-Konsole ein! - Was beobachten Sie und wie lässt sich dies erklären?

```
cat <<ENDE_EINLESEN
Heute ist `date`,
Sie befinden Sie im Verzeichnis `pwd`. Ihr
aktuelles Terminal ist `echo -n $TERM` und
Ihr Heimatverzeichnis finden Sie unter dem Pfad: $HOME.
ENDE_EINLESEN
```

Text 3: Bash-Befehle als inline-Eingabe

Was beobachten Sie und warum beobachten Sie dieses?

3. Aufgabe

Bauen Sie dem „HERE“-Prinzip folgend einen Taschenrechner für die Konsole, indem Sie nachfolgenden Code verwenden:

```
#!/bin/bash
## Konsolen-Taschenrechner
if [ $#  ==  0 ]
then
   echo "Sie haben $0 ohne die zusätzlich benötigte Rechenaufgabe gestartet!"
   exit 1
fi

# Option -l für die mathematische Bibliothek
bc -l <<CALC
$@
CALC
```

Text 4: Skript: rechne

4. Aufgabe

4.1 Beschreiben Sie die Funktion von nebenstehendem bash-Skript!

4.2 Welcher Unterschied ergibt sich, wenn die 7. Zeile nicht „**done <<TEXT**“, sondern „**done <TEXT**“ lauten würde?

```
#!/bin/bash
i=1
while read line
do
   echo "$i. Zeile: $line"
   ((i=$i + 1))
done  <<TEXT
Eine Zeile
`date`
Homeverzeichnis $HOME
Das Ende
TEXT
```

Code 5: Der read-Befehle und die HERE-Technik

Lösung zu 4.1 und 4.2:

5. Aufgabe

Wie setzt man den „IFS“ sinvoll in bash-Skripten ein? - Hierzu zunächst ein einfaches Beispiel:

Beschreiben Sie die Funktion des IFS in nebenstehendem Beispiel!

```
#!/bin/bash
# die voreingestellten Zeichen für IFS werden zunächst gesichert!
BACKIFS="$IFS"
# Minuszeichen als Trenner
IFS=:
if   [  $#  -lt  1  ]
then
    echo "Das Skript: $0 benötigt einen User-login-Namen."
    exit 1
fi
# Ausgabe anhand von Trennzeichen in IFS auftrennen
set `grep ^$1 /etc/passwd`
echo "User            : $1"
echo "User-Nummer     : $3"
echo "Gruppen-Nummer  : $4"
echo "Home-Verzeichnis : $6"
echo "Start-Shell     : $7"
IFS=$BACKIFS
```

Text 6: Skript mit der Anwendung des IFS

er IFS „

6. Aufgabe

In nachfolgendem Skript (Text 7) werden unterschiedliche Techniken und Befehl in Verbindung mit dem IFS angewandt. Beachten Sie den Unterschied zwischen einem **assoziativ adressierten** und **indizierten Array**. Der Befehl **mapfile** generiert gebunden an den Zeilen-Nr. der Text-Datei ein Array. Ein Index-Bezug im Skript sollte immer von Anführungszeichen umrahmt sein (siehe z.B. “${!adresseni[@]}“), ebenso sollte die Ausgabe des Arrays ebenfalls von Anführungszeichen umgeben sein.
Beschreiben Sie die Funktion des nachfolgenden Skripts!

```
#!/bin/bash
Adressena=()            # assoziatives Array oder gehashtes Array
declare -a adresseni    # indiziertes Array oder Array mit nummerischen Index

if [  $#  -lt  1  ] ||  [  !  -f  $1  ]
then
     Skript=`echo "$0" | sed "s/\///g"`
     echo "###### Achtung #########“
     echo "  Das Skript: $Skript muss "
     echo "  mit einer Adressliste.csv aufgerufen werden!"
     ech######################"
     exit 1
fi                                  #cat "$1" | ( mapfile; echo "${MAPFILE[@]}")
mapfile -t < $1             # printf "%s\n" "${MAPFILE[@]}"
mapfile -t < $1
echo "#########--- print mapfile ---################"
printf "%s" "${MAPFILE[@]}"
    # Adressen=`shuf -e "${MAPFILE[@]}"`
Adressena=( "${MAPFILE[@]}" )
adresseni=( "${MAPFILE[@]}" )
    # wichtig sind die Anführungszeichen!
    # --> Siehe hierzu https://sysware.computer/linux/scripte_variablen_arrays.html
echo -e "\n--------------Adressen[@]----------------------------"
    #printf "%s\n" "${Adressen[@]}"
echo "${Adressena[@]}"
echo "#############-----Adressen[2]------------##########"
echo "${adresseni[2]}"
echo "########################################"
declare -p
echo "-#.-#-##--#-#-#-##-##--#-#-#-#"
echo "Das Array adressini[]:"
for INDEX in "${!adresseni[@]}"
do
   echo "Index=\"${INDEX}\" Wert=\"${adresseni[$INDEX]}\""
done
echo "#+#+#+#+#"
adresseni=(  "${adresseni[@]}" )
for INDEX in "${!adresseni[@]}"
do
   echo "Index=\"${INDEX}\" Wert=\"${adresseni[$INDEX]}\""
done
echo "-jetzt würfeln wir die Zeilen neu aus-"
shuf -e  "${adresseni[@]}"
echo "----------Der Index bleibt aber bestehen!-----------"
z=0
while [ $z -lt ${#adresseni[@]} ]
do
  echo "$z -> ${adresseni[$z]}"
  ((z++))
done
echo "----Formatierte Ausgabe der Adressliste ----------"
for INDEX in "${!adresseni[@]}"
do
    #echo "Index=\"${INDEX}\" Wert=\"${adresseni[$INDEX]}\""
    IFSBACK="$IFS"
    Zeileninhalt="${adresseni[$INDEX]}"
    IFS=","
 set $Zeileninhalt
 IFS="$IFSBACK"
 echo -e "\n--- Id: ${INDEX} ----------------------\nName:    $1"
 echo -e "Vorname: $2 \ngeb. am: $3\nStraße:  $4 $5\nPLZ:     $6"
 echo -e "Ort:     $7\nTelefon: $8"
done
```

Text 7: Konsolen-bash-Adressverwaltung

Fortsetzung der Lösung von Aufgabe 6:

```
--- Id: 0 ----------------------
Name:    Meier
Vorname: Peter
geb. am: 03.05.1999
Straße:  Hofgarten 33
PLZ:     91555
Ort:     Ansbach
Telefon: 09123-88842

--- Id: 1 ----------------------
Name:    Huber
Vorname: Rudi
geb. am: 03.06.2000
Straße:  Hufgasse 3
PLZ:     91122
Ort:     Schwabach
Telefon: 09122-82441

--- Id: 2 ----------------------
Name:    Schmid
Vorname: Hans
geb. am: 02.07.1998
Straße:  Zu den Gründen 1
PLZ:     95355
Ort:     Röttenbach
Telefon: 0911-842
```

7. Aufgabe

Testen Sie die beiden Befehle: “echo $TERM“

und “infocmp“!

Welche Ausgaben erhalten Sie?

8. Aufgabe

Sie wollen ein Passwort unsichtbar eingeben. Hierzu verwenden Sie “stty“. Welche Befehlsfolge schaltet die Darstellung der Eingabe ab und welche wieder an?

9. Aufgabe

Der zentrale Befehl zur Umleitung der Eingabe lautet **exec**.

Zeigen Sie, wie sich der Befehl **exec** hinsichtlich ***stdout, stdin*** und ***stderr*** *anwenden lässt*. - Welche Funktion hat die Befehlszeile: ***„Kommando“ >&fd*** bzw. ***„Kommando“ >>&fd.***

10. Aufgabe

Schreiben Sie ein ARRAY mit oder ohne Index in eine Datei. Beginnen Sie Ihr Skript mit ***declare -a Adressen*** oder ***declare -A Adressen!***

Text 8: Array-Werte in eine Datei schreiben

11. Aufgabe

Die Standardfiledeskriptoren: ***stdout "1>", stdin "<0"*** und ***stderr "2>"***sind mit dem Eingabe-Terminal (TTY) verbunden. Neben diesen Standardstreams kann man zusätzliche Filedescriptoren mit den Nummern 3 bis 9 für weitere Umleitungen verwenden. Erläutern Sie den nachfolgenden Code, den man auf einem Terminal eingibt.

```
your@host>  exec   3>  `tty`
your@host>  echo  "Hallo neuer Kanal"  >&3
Hallo neuer Kanal
your@host> exec   3>&-
your@host>  echo  "Hallo neuer Kanal"  >&3
ungültiger Dateideskriptor
```

Text 9: Ein- und Ausgabebeispiel auf dem Terminal

12. Aufgabe

Erläutern Sie nachfolgendes Beispiel:

```
your@host>  w  >  user.dat
your@host>  exec  3<  user.dat
your@host>  read user1  <&3
your@host>  read user2  <&3
your@host>  echo  $user1
trebor tty2 Juni 6  14:05
your@host>  echo  $user2
tot :0 Juni 5  18:09 (console)
your@host> exec   3>&-
```

Text 10: Arbeiten mit dem Filedescriptor

13. Aufgabe

Der besondere Filedescriptoraufruf “<>“!

Eine Funktionsbeschreibung des Skriptes:

```
#!/bin/bash
#         Aufruf mit Parameter
exec     3<>    $1
while  read  line   <&3
do
  echo   $line
  printf  "Hier nach dieser Zeile den neuen
Text für diese Zeile einfügen? [j/n] : "
  read
  [  "$REPLY"  =  "j"  ]  &&  break
done

printf "Bitte hier die neue Zeile eingeben : "
read
echo  $REPLY  >&3
exec 3>&-
exec 3<&-
```

Text 11: Arbeiten mit Filedescriptor-verwendung in zwei Richtungen

Zusatzaufgabe: Ändern Sie das Skript so ab, dass auch die erste Zeile der übergebenen Text-Datei mit neuem Inhalt überschrieben werden kann!

14. Aufgabe

Eine ***„Named Pipe“*** ist ein Konstrukt, das die Ausgabe eines laufenden Prozesses als Eingabe eines anderen Prozesses zur Verfügung stellt (Grundfunktion der Pipe: „|“). Das Besondere an der Named Pipe ist, dass sie nicht wie die Pipe nur mit Bezug zum selben Elternprozess verwendet werden kann, sondern wie eine Datei angesprechbar ist. Die „Named Pipe“ wird von systemnahen Prozessen „Siehe /dev“ sehr häufig als “fifo“-Stream verwendet.

```
your@host>  mkfifo NamedP
your@host>  echo “Hallo lieber Benutzer“ > NamedP
```

*Text 12: Arbeiten mit einer Named Pipe in **Terminal A***

```
your@host>  tail -f  NamedP
Hallo lieber Benutzer
```

*Text 13: Auslesen der Named Pipe in **Terminal B***

An Stelle von mkfifo *Pipename* kann auch der Befehl: mknod *Pipename* p zum Erstellen einer Named Pipe verwendet werden. Denken Sie auch immer daran, wenn unterschiedliche User auf die Named Pipe zugreifen möchten, dass die User die

erforderlichen Zugriffsrechte besitzen. Dennoch können Named Pipes „gesperrt“ sein! – Warum dies so ist, dazu informieren Sie sich bitte auf den nachfolgenden Links und beschreiben Sie anschließend kurz diesen Sachverhalt.

Hier die Links:

https://unix.stackexchange.com/questions/53766/why-mkfifo-behaves-like-a-lifo

https://stackoverflow.com/questions/4113986/example-of-using-named-pipes-in-linux-bash

Beschreibung:

Hier zwei Skripte, die aufeinander bezogen sind und die Sie mit „netcat“ netzwerkfähig erweitern können!

```
#!/bin/bash
## schreibe in die Named Pipe

while true
do
   echo "Mein Text für die Named Pipe"
   echo "Die zweite Zeile noch dazu"
   echo "Und noch eine dritte Zeile"
   break
done > NamedP
```

Text 15: Arbeiten mit Named Pipes, hier der Daten-Schreiber

```
#!/bin/bash
## lese die Named Pipe aus
#
while read zeile
do
   echo $zeile
done < NamedP
```

Text 14: Arbeiten mit Named Pipes, hier der Daten-Verwerter (Empfänger)

Wie verwenden Sie die beiden Skripte?