Videregående shellprogrammering

Innhold

- Input og output til og fra løkker
- Tabeller / arrays
- Mer om håndtering av tekststrenger
- Tomme strenger og defaultverdier *
- Shellfunksjoner
- Håndtering av signaler fra OS *
- Tidsbestemt kjøring av shellprogrammer *
- Debugging og testing *

*: *Ikke* omtalt i kapittel 7 i læreboken

Løkker og input / output

- Vi kan redirigere I/O for løkker i Linux, akkurat som for andre kommandoer

- All lesing fra `stdin` og skriving til `stdout` som foregår *inne i løkken*, mellom nøkkel-ordene `do` og `done` , kan redirigeres til/fra en pipe eller en fil

Eksempel: `while read` (1)

- Kommandoen `read` kan brukes sammen med en `while` - løkke for å behandle input linje for linje:

```
#!/bin/bash
SUM=0
while read X Y
do
  SUM=$((SUM+X*Y))
done
echo $SUM
```

- Én linje med data leses fra `stdin` i hvert gjennomløp av løkken
- *Eller* fra en fil hvis I/O for *hele* scriptet redirigeres

Eksempel: `while read` (2)

```
#!/bin/bash
SUM=0
while read X Y
do
  SUM=$((SUM+X*Y))
done < data.txt
echo $SUM
```

- Her leses dataene i løkken i stedet fra filen `data.txt`
- Hvert gjennomløp leser én linje fra filen, inntil `read` returnerer med exit status ikke lik 0
- Løkken terminerer når slutten på filen er nådd (`end-of-file` er lest)

Eksempel: Nummerere linjer i en fil

```
#!/bin/bash
[ $# -ne 1 ] &&\
{ echo "usage: $0 file"; exit 1; }
[ ! -r $1 ] &&\
{ echo "$0: Unable to read file $1"; exit 1; }

nummer=0
cat $1 | while read linje
do
  ((nummer++))
  echo $nummer: $linje
done > ${1}_nummerert
```

Eksempel: Sortere første linje på alle tekstfiler i stående katalog

```
for file in *.txt
do
   head -1 $file
done | sort -o sorted_headers
```

Array (tabell) i shellprogrammer

- En *datastruktur* som kan lagre flere verdier
- Verdiene ligger 'etter hverandre' i en *indeksert* liste
- Arrayer i Bash indekseres fra 0
- Bash tilbyr bare éndimensjonale arrays
- Hvert element i en array er en shellvariabel – en array kan derfor lagre *både* strenger og tall *

*: I de fleste andre programmeringsspråk er arrayer *homogene* datastrukturer som bare kan inneholde én datatype

Opprettelse av en array

- To ulike metoder for å opprette en tom array `A` med lengde lik 0 (null):

```
declare -a A
A=()
```

- Angir *ikke* lengden på en array ved opprettelse

- Bash vil i stedet sette av plassen som trengs til å lagre dataene, etter hvert som arrayen fylles opp

Ulike metoder for initialisering en array

```
declare -a A1=(Eple Appelsin Pære 3.14)

A2=(Jan Per Ole Kari)

A3=([0]=Jan [1]=Per [2]=Ole [3]=Kari)

A4=([1]=Per [3]=Kari [2]=Ole [0]=Jan)

read -a A5

A6=($(ls))

A7=({a..z} {A..Z} {0..9})
```

Tilgang til elementene i en array A

- Legge inn en verdi: `A[index]=value`
- Hente ut en verdi: `a=${A[index]}`
- Alle elementene: `${A[@]}` eller `${A[*]}`
- Lengde av array * : `${#A[@]}` eller `${#A[*]}`

*: Lengden av en array er definert som antall elementer som ikke er tomme

Eksempel: Utskrift av array

```
declare -a A=(En To Tre Fire Fem)
echo "A[0]=${A[0]}, A[3]=${A[3]}"
echo "Antall elementer: ${#A[@]}"
echo ${A[*]}
echo ${A[@]}
# : kan brukes til å hente delarray
echo ${A[@]:1:3}
# Kommandoen printf tilbyr formatering
# av utskrift, som i progr.språket C
printf "%s\n" "${A[@]}"
```

Eksempel:
Gjennomgang av array med `for` - løkke

```
declare -a A=(Rick Levon Garth Richard Robbie)

for a in ${A[@]}
do
 echo -e "$a\t(inneholder ${#a} tegn)"
done

n=${#A[@]}
for i in `seq 0 1 $((n-1))`
do
 echo "A[$i]=${A[$i]}"
done
```

Eksempel:
Skrive ut array med `while` og `until`

```
declare -a A=(Rick Levon Garth Richard Robbie)

i=0
while [ i -lt ${#A[@]} ]
do
 echo "A[$i]=${A[$i]}"
 ((i++))
done

i=${#A[@]}
until [ i -eq 0 ]
do
 ((i--))
 echo "A[$i]=${A[$i]}"
done
```

Eksempel:
Alle filer i et katalogtre med file mode 755

```
[ $# -ne 1 ] && { echo usage: $0 directory; exit 1; }
[ -d $1 ] || { echo $0: $1 not a directory; exit 1; }

# Legger alle vanlige filer i katalogtreet i en array
temp=($(find $1 -type f))

# Skriver ut navnet på filer med mode 755
for file in "${temp[@]}"
do
  if ls -l $file | egrep -q 'rwxr\-xr\-x'
  then
    # Fjerner path før utskrift av selve filnavnet
    basename $file
  fi
done
```

Mer om håndtering av tegnstrenger

- Har tidligere sett enkel strenghåndtering med `expr`
- Bash tilbyr en egen syntaks for å gjøre streng-operasjoner på skallvariabler
- Syntaks:

  `${`*`variabel`*`[`*`operasjon(er)`*`]}`

- Dette er egentlig bare en syntaktisk "shortcut":
  - Bash oversetter egen syntaks til `expr` – syntaks
  - `expr` utfører håndteringen av strenger "bak kulissene"

Noen Bash-operasjoner på strenger

`${#navn}`	Antall tegn i skallvariabelen `navn`
`${navn:3}`	Substreng fra 4. tegn (0 er første tegn)
`${navn:2:3}`	Substreng fra 3. til 5. tegn
`${navn/en/an}`	Bytt *første* forekomst av en med an
`${navn//en/an}`	Bytt *alle* forekomster av en med an
`${navn//en/}`	*Fjern* alle forekomster av en
`${navn/#en/an}`	Bytt bare hvis en er i *starten* på streng
`${navn/%en/an}`	Bytt bare hvis en er i *slutten* av streng

Eksempel: .html → .htm

```
#!/bin/bash
[ $# -ne 1 ] &&\
{ echo usage: $0 directory; exit 1; }
[ -d $1 ] ||\
{ echo $0: $1 not a directory; exit 1; }

cd $1
for filename in $(ls *.html 2> /dev/null)
do
   if [ -e ${filename/%html/htm} ]
   then
      echo "${filename/%html/htm} already exists"
   else
      mv $filename ${filename/%html/htm}
   fi
done
```

Fjerne deler av tegnstreng med regex-matching

#	Fjern korteste match fra starten av streng
##	Fjern lengste match fra starten av streng
%	Fjern korteste match fra slutten på streng
%%	Fjern lengste match fra slutten på streng

```
foo="this is a test"
${foo#t*is}    is a test
${foo##t*is}   a test
${foo%t*st}    this is a
${foo%%t*st}
```

Ny utgave av `.html` → `.htm`

```
[ $# -ne 1 ] &&\
{ echo usage: $0 directory; exit 1; }
[ -d $1 ] ||\
{ echo $0: $1 not a directory; exit 1; }

cd $1
for filename in $(ls *.html 2> /dev/null)
do
   if [ -e ${filename/%l} ]
   then
      echo "${filename/%l} already exists"
   else
      mv $filename ${filename/%l}
   fi
done
```

Tomme variabler og defaultverdier

- En *tom* variabel inneholder bare verdien NULL
- Bash tilbyr mulighet til å bruke en *alternativ* defaultverdi hvis en variabel er (eller ikke er) tom
- Defaultverdier kan anvendes til:
  - Fleksibel håndtering av parametre til shellscript
  - Mulighet for default svar ved interaktiv input

Operatorer for default-håndtering

`${variabel:-streng}`	Bruk verdien `streng` hvis `variabel` er tom
`${variabel:=streng}`	Bruk verdien `streng` hvis `variabel` er tom, og sett `variabel` lik `streng`
`${variabel:+streng}`	Bruk verdien `streng` hvis `variabel` *ikke* er tom
`${variabel:?melding}`	Skriv ut (feil)melding hvis `variabel` er tom

Eksempler: Defaultverdier

Defaultverdier for innparametre:

```
INFILE=${1:-"infile.txt"}
OUTFILE=${2:-"outfile.txt"}
sort $INFILE > $OUTFILE
```

Defaultverdi i innlesning av verdi fra bruker:

```
read -t 10 navn
navn=${navn:="Levon Helm"}
echo "Hællæ, $navn"
```

Shellfunksjoner

- Tilsvarer *metoder** i språk som f.eks. Java:
  - Samler flere kommandoer i en *gruppe* eller *kodeblokk*
  - Kommandoene kan da senere brukes bare ved å referere til shellfunksjonens *navn*
  - En shellfunksjon kjøres som en vanlig kommando i shellet, men starter *ikke* noen ny prosess
- Brukes til:
  - Gjenbruk og modularisering av kode
  - Lage *shortcuts* for lengre, kompliserte kommandoer

*: Aka «funksjoner», «prosedyrer», «subrutiner»

Syntaks for å deklarere shellfunksjoner

```
[function] funksjonsnavn() {
   kommando
   kommando
   ...
}
```

- Nøkkelordet `function` er valgfritt og kan sløyfes
- Det skal ikke stå noe mellom `()`
- `()` kan sløyfes hvis nøkkelordet `function` brukes

Syntaksvarianter

```
function kl() {
  echo "$(date | cut -d' ' -f4)"
}

kl() {
  echo "$(date | cut -d' ' -f4)"
}

function kl {
  echo "$(date | cut -d' ' -f4)"
}

kl() { echo "$(date | cut -d' ' -f4)"; }
```

Bruk av shellfunksjoner

- Funksjoner må alltid defineres før de kan brukes
- Kalles ved bare å angi funksjonsnavnet
- Funksjoner kan ta i mot innparametre:
  - Fungerer som for et vanlig shellscript
  - Listes opp etter funksjonsnavnet
  - Refereres til som `$1`, `$2`, `$3` etc.
- Se koden til en funksjon: `type funksjonsnavn`

Eksempler: Shortcuts

```
function lsext() {
  # Long listing of files with given extension
  EXT=$1
  ls -l | egrep "^-.*\.$EXT$"
}
```

```
function rpass() {
  # Create random strong alphanumeric
  # password w/given lenght (default 12)
  LENGTH=${1:-12}
  cat /dev/urandom | tr -cd '[:alnum:]' |\
  head -c $LENGTH; echo
}
```

En funksjon med lokal variabel og retur-verdi *

```
#!/bin/bash
max( ) {
  [ $# -lt 1 ] && { echo "$0: No input"; exit 1; }

  local m=$1

  shift
  for n in $@
  do
    if [ $n -gt $m ]
    then
      m=$n
    fi
  done

  return $m
}
```

*: Funksjonen virker bare for heltall i intervallet [0, 255], det som returneres er egentlig en «exit-status»

Eksempel: Gjenbruk av kode

```
function start_tag ()
{
  echo "<$1>"
}

function end_tag ()
{
  echo "</$1>"
}

function tag_txt () {
  start_tag $1
  echo $2
  end_tag $1
}
```

```
start_tag HTML
start_tag HEAD
tag_txt TITLE "Jan sin\
hjemmeside"
end_tag HEAD
start_tag BODY
tag_txt H1 "Get lost"
tag_txt P "The internet\
is full, go away"
end_tag BODY
end_tag HTML
```

`trap` – håndtering av signaler i script

- Operativsystemet kan sende signaler til prosesser, ofte for å signalisere at noe er 'galt', f.eks.:
  - Divisjon med null
  - Bruker har trykket interrupt (`Ctrl-C`)
  - Et device har feilet... og mye annet
- Signaler får ofte OS til å terminere prosessen
- Vi kan bruke kommandoen `trap` til å håndtere signaler inne i shellprogrammer, for f.eks. å terminere kontrollert hvis noe går galt

Bruk av `trap`

`trap [`*`kommandoer`*`] `*`signalkode`*`[`*`r`*`]`

- Når et av signalene som er listet opp mottas, utføres *`kommandoer`* i stedet for default signalhåndtering i OS
- Oversikt over alle signalkoder: `trap -l`
- Kan også angi noen andre signalkoder til `trap` , bl.a.:

EXIT	kode utføres når shellprogramet terminerer
ERR	kode utføres når kommandoer feiler

- Eksempel:

```
trap 'rm -f /tmp/xyz$$; exit' HUP ALRM ERR EXIT
```

Det plagsomme scriptet

```
#!/bin/sh

trap 'echo "He-he, det hjelper ikke med ^C"' INT

while [ true ]
do
   echo -n "Jeg er shellscriptet som skal "
   echo "plage deg til evig tid >:-)"
   sleep 3
done
```

`sleep` og `at` – Tidsbestemt kjøring

- **`sleep`** : Vent en bestemt tid før neste kommando

```
{ sleep 2h; sort BIGfile > SORTEDfile } &
```

- **`at`** : Start shellscript på et bestemt tidspunkt

```
at 4:30 am tomorrow < shellprogram
at now + 3 days < shellprogram
at 11:00am Monday < shellprogram
at 5:00pm Feb 15 2015 < shellprogram
```

- **`atq`** : Se køen av ventende kobber
- **`atrm`** : Fjerne jobb fra køen

Et shellprogram som kjører seg selv en gang i uken

```
#!/bin/bash

if who | grep -q janh
then
  echo "Har ikke du fått sparken ennå?"
fi
cat $0 | at now + 1 week
```

Debugging og testing av shellprogrammer

- Shellprogrammering krever prøving og feiling
- Test interaktivt hver ny programlinje som skrives
- Debuggingsopsjonene `-v` og `-x`:

`bash -v program`	Skriver ut kodelinjene etterhvert som de utføres
`bash -x program`	Skriver ut kommandoer som utføres og verdien av variable som brukes