

# Lektionstillfälle 11

Skriptning 2  
med Mikael Larsson

# Återblick

Förra gången började vi skripta i Bash.

Vi arbetade med variabler, argument, variabelexpansion, if-satser och tester.

# Dagens lektion

**Mål: Kunna skapa skript med iteration och funktioner.**

- Funktioner och lokala variabler
- Loopar med for och while
- Beräkningar
- Inläsning med while read

**TLCL Kapitel 26, 29, 33, s482-487**

# Inlämningsuppgifter och tenta

Kursens examinationsgrundande moment är:

- Inlämningsuppgift 1
- Inlämningsuppgift 2
- Tentamen

För att klara kursen krävs betyget godkänd (G) på alla tre moment.

För alla tre moment är kravet på G 60%.  
Kravet för VG på tentamen är 80%.



# Termer och begrepp

- local
- while
- for
- read
- IFS – variabeln

# Shell-funktioner

Vi berörde shell-funktioner när vi pratade om olika slags kommandon i lektion 3.

Shell-funktioner är ett sätt att dela upp sina skript i mindre delar, precis som funktioner i andra språk.

En stor skillnad är att shell-funktioner inte har returvärdet som man är van vid. De fungerar som exit-codes och måste vara numeriska.

Man kan inte heller deklarerera funktionsargument, utan de fungerar som skript-argument och tilldelas \$1, \$2, et.c.

```
#!/bin/bash

minfunktion(){
    return 42
}

minfunktion

echo $?
```

# Lokala variabler

Variabler som används i en funktion delas med hela skriptet, om de inte deklareras med "local".

Använd som vana lokala variabler, för att undvika nedskräpning i det globala kontextet.

```
#!/bin/bash

minfunktion(){
    meddelande="hej"
    local lokal=1066
}

minfunktion

echo $meddelande $lokal
```

# while

“while” fungerar som i andra språk, och villkoret, eller testet, fungerar precis som för “if”:

```
#!/bin/bash

echo -n Waiting

while true; do
    echo -n .
    sleep 1
done

echo "This never happens"
```



# while..

Ett mer användbart? exempel:

```
#!/bin/bash

while (($((date +%S) % 10 != 0)); do
    echo waiting
    sleep 1
done

echo "It's finally time!"
```

# Switch statement - case \$var in esac

Switch statement heter case i bash och avslutas med esac.  
Vid många olika värden blir det elegantare än **if elif else fi**:

```
while [ $# -gt 0 ]; do
    case $1 in
        build)
            echo "build command"
            ;;
        run)
            echo "run command"
            ;;
        -*)
            option=$1
            shift
            echo "option '$option' found with value '$1'"
            ;;
    esac
    shift
done
```

# for

“for”-loopar finns i två olika varianter; shell-style och C-style.

Shell – varianten arbetar på ett argument som expanderas till en lista. Loop-variabeln tilldelas ett element i listan för varje varv:

```
for v in 1 2 3 4; do
    echo $v
done

for f in *.sh; do
    ls $f
done

for z in {A..D}{1..3}; do
    echo $z
done
```



# C-style for

Det är mindre vanligt att man ser den här sortens for-loop i bash, men den fungerar precis som C-loopar, och ser ut såhär:

```
for (( i=0; i<5; i=i+1 )); do  
    echo $i  
done
```



# Avancerad for loop över array-index

I programspråk är det vanligt att loopa över array-index och det kan man naturligtvis även göra i bash.

```
#!/bin/bash

myarray=(apple banana lemon orange);
for index in ${!myarray[@]}; do
    echo "$index: ${myarray[$index]}";
done

echo "index från array: ${!myarray[@]}"
echo "värde från 2:a elementet: ${myarray[1]}"
echo "arrayer är (zero based)"
```

# Beräkningar

Vi började titta på beräkningar förra lektionen, genom aritmetisk expansion:

```
echo $((42 * 2))      (bara heltal)
```

Det finns lite mer notation och flera numeriska operatorer som kan vara bra att känna till.

Man kan skriva tal i olika baser:

```
$((255))  
$((10#255))  
$((0xff))  
$((16#ff))  
$((2#11111111))
```

Med bc kan man gå åt andra hållet:

```
echo "255" | bc  
echo "obase=10; 255" | bc  
echo "obase=16; 255" | bc  
echo "obase=16; 255" | bc  
echo "obase=2; 255" | bc
```

```
$((10#8 + 2#111 + 16#F0))
```

 (Det går att blanda talbaser i en beräkning)

Med bc kan man räkna med decimaler: `echo "scale=2; 255/7" | bc`

# Numeriska operatorer

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Integer division
**	Exponentiation
%	Modulo (remainder)



# Tilldelningsoperatorer

Notation	Description
<code>parameter = value</code>	Simple assignment. Assigns <i>value</i> to <i>parameter</i> .
<code>parameter += value</code>	Addition. Equivalent to <code>parameter = parameter + value</code> .
<code>parameter -= value</code>	Subtraction. Equivalent to <code>parameter = parameter - value</code> .
<code>parameter *= value</code>	Multiplication. Equivalent to <code>parameter = parameter * value</code> .
<code>parameter /= value</code>	Integer division. Equivalent to <code>parameter = parameter / value</code> .
<code>parameter %= value</code>	Modulo. Equivalent to <code>parameter = parameter % value</code> .
<code>parameter++</code>	Variable post-increment. Equivalent to <code>parameter = parameter + 1</code> (however, see the following

Detta är klippt från The Linux Command Line sida 485



# Laboration 1

Se instruktion i portalen!

# Läsa från filer med "for"

För att läsa in rader från en fil skulle vi kunna göra såhär:

```
for x in $(cat ../man.txt) ; do
    echo $x
done
```

Det finns ett problem här. "for" tar ju en lista som argument, och konverteringen till en lista delar upp varje ord i filen.

För att undvika detta kan man sätta IFS – variabeln till "\n":

IFS=\$'\n'

*Observera att om du inte återställer IFS kan andra kommandon ge oönskade resultat.*

**IFS=\$' \t\n'**

*IFS = Internal Field Separator, används för att dela upp fält vid konvertering till lista.*

*"\n" = newline - tecknet*

***OBS att flera upprepade IFS – tecken räknas som ett. I exemplet här kommer alltså tomma rader tas bort!***

# Läsa från stdin med while read

Det är vanligare att man läser in filer med “read”:

```
while read line; do
    echo $line
    break
done < ../man.txt
```

Här finns en annan utmaning. “read” läser en rad i taget från stdin, och gör sen samma konvertering till lista som i “for” – fallet.

Sen tilldelas varje element i listan till argumenten till read. Alla kvarvarande element tilldelas det sista argumentet, som här är “line”.

Så “line” får hela radens värde, men dubletter av IFS elimineras.

*Här är lösningen – om man vill behålla raderna obehandlade – att sätta IFS till tom sträng:*

**IFS=' '** *I detta fall fungerar även* **IFS=\$'\n'**



# Läsa stdout från kommandon med while read

Det finns två sätt att omdirigera stdout från en **process** in till **while read**, från höger och från vänster:

```
while read line; do
    echo $line
    break
done < <(cat ../man.txt)

cat ../man.txt | while read line; do
    echo $line
    break
done
```

I det första fallet används en “pipe” – variant vi inte sett tidigare.

I det andra fallet används en vanlig “pipe”.  
Resultatet här är identiskt, men det finns en skillnad – den andra while-loopen körs i en ny process!

*P.g.a. de konstigheter som kan uppstå när loopen är i en ny process används nästan alltid den första varianten för **while read** – loopar.  
Vad kan konstigheterna bero på?*



# while read – exempel

Konstigheterna till trots, så är “while read” mycket användbart:

```
#!/bin/bash

query=${1:-rad}
lines=0
word_count=0

while read line; do
    lines=$((lines+1))
    for word in $line; do
        if [ "$word" == "${query}" ]; then
            word_count=$((word_count+1))
        fi
    done
done < Public/four_lines.txt

echo "Searched $lines lines and found $word_count instances of the word '${query}'"
```

# while read – ett till exempel

```
#!/bin/bash

IFS=":"

while read user _ _ _ _ home shell; do
    if [[ $shell == *nologin ]]; then continue; fi
    echo "User $user lives in $home and uses shell $shell"
done < /etc/passwd
```

*Understreck används för att skippa vissa fält.*

# Laboration 2

Se instruktion i portalen!



# Summerring

Idag har vi skriptat lite mer!

Vi har använt shell-funktioner, loopat, läst från filer och gjort lite beräkningar.



# Nästa gång

**Mål: Att kunna hitta, hantera och söka i loggfiler.  
Förstå och skapa reguljära uttryck.**

- Loggfiler, less och tail
- Textbehandling: grep, sed, cut, sort, uniq
- Reguljära uttryck
- **Inlämningsuppgift 2**

**Stort tack!**