Intro Linux, C, PowerShell
Erik Hjelmås, 22. august 2026

Innhold

1 Komme i gang

Lag din egen virtuelle Linux-maskin (Linux-VM", Linux-host", Linux-maskineller bare Linux- vi bruker disse betegnelsene om hverandre) på din egen bærbare PC ved å følge instruksjonene: How to install Linux on Windows with WSL. Installer den nyeste Ubuntu LTS-versjonen (LTS står for Long-Term Support) som i skrivende stund er Ubuntu 26.04 (26.04 fordi den ble gitt ut i april (den fjerde måneden) i 2026). Etter at du har installert Linux, har du to måter å nå den på, og du må beherske begge:

Åpne en terminal Start programmet "Ubuntufra Windows-menyen.

Åpne en ssh-forbindelse For å få til dette må du først installere en ssh-tjener inne i Linux-VM-en:

1. Nå Linux ved å åpne en terminal som beskrevet over. Kjør følgende kommandoer:

   ```
   sudo apt update
   sudo apt install openssh-server
   ```

2. Kjør kommandoen `ip address` for å se alle IP-adressene som finnes på ulike grensesnitt i din Linux. Du skal se en IP-adresse som starter med 172.16 – skriv den ned.
3. Fra den bærbare Windows-PC-en kan du nå åpne PowerShell og skrive `ssh ubuntu@YOUR_IP_ADDRESS` der du erstatter `YOUR_IP_ADDRESS` med 172.16-adressen du fant (kanskje du også bare kan skrive `ssh ubuntu@localhost`).

Hver gang du ser en oppgave på de følgende sidene som refererer til ssh, skal du nå din Linux med ssh før du gjør den oppgaven.

2 Why Bother?

Hvorfor bry seg med kommandolinjen når det finnes grafiske grensesnitt overalt? Fordi nesten alt du skal gjøre som cyberingeniør skjer nettopp her: undersøke en mistenkelig maskin, lete gjennom logger, automatisere en oppgave, eller fjernstyre en tjener du bare når over ssh. En angriper som har brutt seg inn på et system har sjelden et skrivebord å klikke på – de har et shell. Skal du forstå og forsvare systemer, må du beherske det samme verktøyet. De neste timene bygger vi denne ferdigheten fra bunnen av.

Vi kommer til å følge denne utmerkede tutorialen Learning the shell"i detalj. Denne tutorialen er sammen med dette heftet ditt pensum, og inneholder alle oppgavene du trenger å gjøre.

2.1 What is "the Shell"?

Les teksten og studer/gjør eksemplene på What is "the Shell"?. Tilleggsnotater fra meg:

- Pass på at du bruker Linux som en vanlig bruker og ikke som root-brukeren (å gjøre noen oppgaver som root og noen som vanlig bruker vil gi deg irriterende tilgangskontroll-problemer senere, stol på meg :)
- Legg merke til hvordan du bruker piltastene for kommandolinjehistorikk, og hvordan du kan kopiere og lime inn tekst (du kan ikke bruke CTRL-C, siden det avslutter et program som kjører i stedet for å kopiere tekst)

Oppgaver:

1. Hva er forskjellen på *et shell* og *en terminal*?
2. Hva heter shellet vi bruker?
3. Hva er den alternative måten å samhandle med datamaskiner på i stedet for å bruke et kommandolinje-shell?

2.2 Navigation

Les teksten og studer/gjør eksemplene på Navigation. Tilleggsnotater fra meg:

- I gamle dager uten smarttelefoner/nettbrett hadde vi bare PC-er og bærbare, og filhierarkiet (trestrukturen) var alltid synlig og lettere å forstå. Filhierarkiet finnes fremdeles på smarttelefonen din, men det er skjult bak alle appene.

Oppgaver:

1. Hva er din *arbeidskatalog* (working directory)?
2. Finnes det noen filer i denne katalogen?
3. Bytt katalog til `.ssh`.
   (merk: hvis denne katalogen ikke finnes så gjør denne kommandoen
   `cd && mkdir .ssh && touch .ssh/authorized_keys`)
4. Hvilke(n) fil(er) finnes i denne katalogen?
5. Bytt katalog til *foreldrekatalogen* (parent directory).
6. Hva er arbeidskatalogen din nå?

7. Bytt katalog til `/home` med *absolutt filsti*.
8. Bytt katalog tilbake til `.ssh`-katalogen.
9. Bytt katalog til `/home` med *relativ filsti*.
10. Bytt katalog til roten av filsystemet.
11. Bytt katalog til hjemmekatalogen din *på tre forskjellige måter*.

2.3 Looking Around

Les teksten og studer/gjør eksemplene på Looking Around. Tilleggsnotater fra meg:

- Legg merke til syntaksen og terminologien
  `command -options arguments`
  (noen opsjoner trenger argumenter, andre gjør det ikke).
- Noen ofte brukte opsjoner til ls er:
  ```
  ls -ltr  # sorter filer etter tid, omvendt rekkefølge
  ls -ltrh # lesbar output (filstørrelse i KB/MB/GB)
  ```

Oppgaver:

1. List alle *skjulte filer* i hjemmekatalogen din.
2. Hva er filnavn, endringstidspunkt, størrelse i byte, gruppe, eier og rettigheter for fila du fant i `.ssh`-katalogen?
3. Vis innholdet i fila `/etc/services` med `less`. Søk etter tegnene `http`, gjenta søket til det står Pattern not found". Avslutt `less`.
4. (merk: denne oppgaven er bare relevant hvis du har brukt ssh for å nå din Linux) Hva slags fil er fila i `.ssh`-katalogen? Forstår du hvordan denne fila henger sammen med en fil du bruker når du kobler deg til din Linux med ssh?

2.4 A Guided Tour

Les teksten og studer/gjør eksemplene på A Guided Tour. Tilleggsnotater fra meg:

- Fra nå av, husk å alltid bruke *TAB-fullføring* for å fylle ut kommandoene og filnavnene/stiene dine.

Oppgaver:

1. Bytt katalog til roten av filsystemet. Bytt til en underkatalog minst fire nivåer under rotkatalogen. Gå tilbake til rotkatalogen og gjenta to ganger til (med forskjellige kataloger hver gang). Målet her er å øve på TAB-fullføring. Hint: start med `/usr/lib`-katalogen.
2. I hvilke kataloger finner du de fleste programmene du kan kjøre?
3. I hvilke kataloger finner du de fleste konfigurasjonsfilene? (dette er filer som inneholder innstillinger som bestemmer oppførselen til systemet og programmene)

2.5 Manipulating Files

Les teksten og studer/gjør eksemplene på Manipulating Files. Tilleggsnotater fra meg:

- Hva er en fil? Det er data (”innholdet”) og metadata (filnavn, filstørrelse, tidsstempler, rettigheter, osv.)

Oppgaver:

1. Kopier fila `/etc/services` til hjemmekatalogen din. Gi den nytt navn `commonports`
2. Bytt katalog til `/tmp`. Lag en katalog `mytmp` i `/tmp`. Flytt fila `commonports` fra forrige oppgave til den nyopprettede `mytmp`-katalogen. Bekreft at fila er der og ikke lenger i hjemmekatalogen din.
3. Lag en katalog `backups`. Kopier katalogen `/etc/terminfo` inkludert alle filer til `backups`-katalogen ved å bruke opsjonen `-a`. Å bruke opsjonen `-a` er en viktig måte å kopiere på som kalles ”arkivmodus: den bevarer all metadata (tidsstempler, rettigheter, eier, gruppe) som brukeren som utfører kopieringen har lov til å bevare. Hvilken metadata ble ikke bevart i denne kopieringen?
4. Bytt katalog til `/usr/bin`. List alle filer med filnavn som:
   (a) starter med en `s`
   (b) slutter med enten en `m` eller en `n`
   (c) bare har to tegn
   (d) inneholder et punktum (`'.'`)
   (e) starter med en `s` etterfulgt av enten `c`, `e` eller `f`, og har ett ekstra tegn av hvilket som helst slag (med andre ord skal filnavnet ha tre tegn totalt).

2.6 Working with Commands

Les teksten og studer/gjør eksemplene på Working with Commands. Tilleggsnotater fra meg:

- Vi trenger å vite dette fordi en kommando noen ganger finnes i flere varianter under samme navn, og vi blir forvirret hvis den oppfører seg annerledes enn forventet når vi bruker feil variant (dette har skjedd meg med `time`-kommandoen, som jeg bruker ganske ofte).

Oppgaver:

1. Hva er de fire forskjellige typene kommandoer?
2. Hva slags kommando er `pwd`? `python3`? `ll`?
3. `time` er en kommando vi kan bruke for å måle kjøretiden til et program. Den finnes som et *Shell keyword*, som er forskjellig fra en *Shell builtin*. En builtin er bare et program kompilert inn som en del av shellet (for bedre ytelse), mens et keyword er en del av shellets språk (ja, et shell er også et programmeringsspråk). `time` finnes også som et eget program med mer funksjonalitet enn keywordet. Hvor i filsystemet ligger `time`-programmet? Hva skjer hvis du spør shellet hva slags kommando `time` er?
4. Bla gjennom hjelpesiden for builtin-en `echo`. Hva betyr hakeparentesene? Prøv å skrive ut litt tekst med `echo` med og uten linjeskift.
5. Vis man-siden til `netcat`-programmet. Søk i denne man-siden etter ordet examples"(husk at søk i man-sider fungerer akkurat som søk i `less`-programmet, der du kan skrive 'n' for neste treff).

2.7 I/O Redirection

Fram til nå har du kjørt én kommando av gangen. Nå kommer det som gjør kommandolinjen virkelig kraftig: du kan koble sammen enkle kommandoer til dine egne verktøy. Vil du vite hvilke prosesser som bruker mest minne, hvor mange filer i et katalogtre som matcher et mønster, eller hvilke tjenere som svarer på nett – så er svaret som regel en kort pipeline av kommandoer du allerede kan. Dette er kjernen i hvordan en analytiker siler gjennom store datamengder raskt.

Les teksten og studer/gjør eksemplene på I/O Redirection. Tilleggsnotater fra meg:

- Viktige kommandoer å beherske er `cat`, `sort`, `uniq`, `head`, `tail`, `grep` og `wc`.

Oppgaver:

1. Lag en fil som bare inneholder tallet `1` med kommandoen
   `echo 1 > test.txt`

   (a) Legg til tallene `2` og `3` på slutten av fila slik at outputen blir

   ```
   $ cat test.txt
   1
   2
   3
   ```

   (b) Lag fila på nytt med bare tallet `1`, men legg nå til tallene `2` og `3` *med to kommandoer men uten linjeskift, og bruk deretter en tredje kommando for å legge til linjeskiftet* (på slutten av fila) slik at outputen blir

   ```
   $ cat test.txt
   1
   23
   ```

2. Last ned en fil med navn:
   `curl -O https://erikhje.folk.ntnu.no/navn.dat`

   (a) Bruk `wc` til å telle antall linjer og ord i fila.
   (b) Sorter innholdet i fila og skriv outputen til en ny fil `snavn.dat`.
   (c) Vis hvert navn i fila bare én gang.
   (d) Tell antall linjer som inneholder `Emil`
   (e) Vis en liste over de hyppigst forekommende navnene slik (hint: `man uniq`):

   ```
   5 Emil
   3 Frida
   2 Per Arne
   2 Emma
   2 Ella
   1 William
   1 Sofie
   etc
   ```

3. Skriv en kommando som viser de fem første linjene i fila `~/.bashrc`.

4. Skriv en kommando-pipeline som skriver linje 21-24 av `~/.bashrc` til en ny fil `strange.tmp`.

5. En vanlig førstesjekk når en maskin er treg eller full, er å finne ut hva som bruker mest diskplass. Kommandoen `du -s *` (disk usage") skriver ut hvor mye plass hver fil og katalog i arbeidskatalogen bruker. Stå i hjemmekatalogen din (hvis du har Windows på laptop'n din så gå gjerne til Windows-hjemmekatalogen din `/mnt/c/Users/BRUKERNAVN`) og skriv en pipeline som viser de tre som bruker mest, med den største øverst, og lagrer resultatet i en fil `diskbruk.txt`.

2.8 Expansion

Les teksten og studer/gjør eksemplene på Expansion. Tilleggsnotater fra meg:

- Fra nettsiden: Each time we type a command line and press the enter key, *bash performs several processes upon the text before it carries out our command*".

Oppgaver:

1. Regn ut 32^3 med aritmetisk ekspansjon.
2. Lag følgende katalogstruktur med `mkdir -p` og krøllparentes-ekspansjon (brace expansion).

   ```
   .
   |--A
   |   |--0
   |   |--1
   |
   |--B
   |   |--0
   |   |--1
   |
   |--C
       |--0
       |--1
   ```

   Bekreft at den er opprettet med `ls`-kommandoen ved å bruke opsjonen for rekursiv listing (rekursivbetyr i denne sammenhengen å vise alle underkataloger). Slett hele katalogstrukturen du lagde med én enkelt kommando.
3. Se alle de definerte *miljøvariablene* med `printenv | less`. Bruk `echo` til å skrive ut verdien av `SSH_CONNECTION` (eller `TERM` hvis `SSH_CONNECTION` ikke finnes).
4. Kjør kommandoen `echo I am $USER`.
   (a) Gjenta denne kommandoen, men erstatt `$USER` med *kommandosubstitusjon* og kommandoen `whoami`.
   (b) Gjenta kommandoen, men legg til anførselstegn slik at outputen blir
   `I am ubuntu`
   (c) Gjenta kommandoen `echo I am $USER`, men legg til anførselstegn slik at outputen blir `I am $USER`

5. Skriv en kommando-pipeline med kommandosubstitusjon som skriver ut antall kataloger i `/usr`. Den skal gi outputen:
   `There are 12 dirs in /usr`

6. Last ned en fil med litt tekst:
   `curl -O https://erikhje.folk.ntnu.no/MC.txt`
   Skriv tre kommandolinjer som skriver antall forekomster av ordene `wish`, `pizza` og `rub` (henholdsvis) i `MC.txt` til en fil `wordstat.dat`. Etter at du har kjørt de tre kommandolinjene skal du kunne gjøre dette

   ```
   $ cat wordstat.dat
   wish: 5
   pizza: 2
   rub: 3
   ```

7. Bruk `echo` med *backslash-escape-tegnene* for linjeskift og tabulator til å produsere følgende output:

   ```
   My test of linebreaks (newline)
   and
       TAB
           and more TABS
   ```

2.9 Permissions

Les teksten og studer/gjør eksemplene på Permissions. Tilleggsnotater fra meg:

- Rettigheter på Linux er faktisk ganske enkle å forstå, i motsetning til rettighetene på Windows…

Oppgaver:

1. Før du starter denne oppgaven, kjør `whoami` for å forsikre deg om at du er `ubuntu`-brukeren, IKKE `root`. Skriv kommandoer for følgende
   (a) Lag en fil `a.txt`
   (b) Endre rettighetene på `a.txt` til `r--r-----`
   (c) Prøv å slette `a.txt` (gjenta stegene over hvis du klarer å slette den)
   (d) Endre rettighetene på `a.txt` til `rw-------`

(e) Legg til en bruker `mysil` (`adduser mysil --disabled-login` men du har ikke tilgang til å gjøre dette, så hva må du legge til i denne kommandoen?).

(f) Endre eieren av `a.txt` til å være `mysil`

(g) Prøv å slette `a.txt` (gjenta stegene over hvis du klarer å slette den)

(h) Flytt `a.txt` til `/tmp/`

(i) Prøv å slette `/tmp/a.txt`

3 Writing Shell Scripts

En kommandolinje du skriver én gang er nyttig; en du kan kjøre om igjen – mot hundre maskiner, eller hver natt klokka tre – er et verktøy. Det er dette skript gir deg: du fanger arbeidet ditt i en fil, gir den argumenter, lar den ta valg med if-tester og gjenta seg med løkker. Både forsvarer og angriper lener seg tungt på skript for å samle inn, overvåke og automatisere. Fra nå av slutter vi å gjøre ting for hånd.

3.1 Writing Our First Script and Getting It to Work

Les teksten og studer/gjør eksemplene på Writing Our First Script and Getting It to Work. Tilleggsnotater fra meg:

- Viktig for deg å forstå miljøvariabelen `PATH`, siden den kan misbrukes til å lure (angripe) brukere til å kjøre *trojanske hest-programmer/skript* (programmer/skript som ser ut til å gjøre noe annet enn det de faktisk gjør).

Oppgaver:

1. En kommentar i et shell-skript starter med en hash (`#`), men et skript[1] starter alltid med de to tegnene `#!`. Dette kalles *shebang* (sannsynligvis fra shell-bang") eller *hashbang*. Hva er spesielt med denne to-tegns- kombinasjonen som alltid er til stede i begynnelsen av et skript? (med andre ord: hva gjør den?)
2. Følg nettsiden Writing Our First Script and Getting It to Work:
   (a) Velg et tekstredigeringsprogram du har lyst til å lære. Hvis du ikke har noen spesiell interesse for dette, anbefaler jeg enten `micro` hvor du kan bruke de fleste hurtigtastene du kanskje kjenner fra teksthandlere som word, eller `nano`, der du kan se en "menynederst hvor f.eks. `^X Exit` betyr å holde nede CTRL-tasten mens du trykker x-tasten – dette lar deg lagre og lukke fila.
   (b) Gjør alle kommandoene på nettsiden for å lage ditt første `hello_world`-skript og legg det som et kjørbart skript i `PATH`-en din.

3.2 Editing the Scripts We Already Have

Dette kapittelet er ikke obligatorisk å lese, det er ikke en del av pensum, men les det hvis du er interessert (med andre ord: denne nettsiden vil ikke være en del av eksamen).

[1] Dette gjelder også skript i andre språk som Python, Perl, osv.

3.3 Here Scripts (aka "Here document")

Les teksten og studer/gjør eksemplene på Here Scripts. Tilleggsnotater fra meg:

- Vi skal følge teksten og lage nettsider ved hjelp av et shell- skript.

Oppgaver:

1. Velg en av følgende to måter[2] å lage nettsider på (løsningene du får er basert på alternativ 1b med en webtjener på din Linux-VM):
   (a) Bruk hjemmeområdet du allerede har på nett gjennom NTNU:
      i. ssh til `login.stud.ntnu.no` med NTNU-brukernavnet og -passordet ditt.
      ii. Legg HTML-filene dine i `public_html`-katalogen.
      iii. Se nettsiden på `https://folk.ntnu.no/USERNAME/FILENAME` (erstatt `USERNAME` og `FILENAME`)
   (b) Installer en webtjener på din Linux-VM og legg nettsidene dine der:

```
# Kjører det noen programmer som lytter på porter?
ss -tlpn # viser sannsynligvis port 22 for ssh og 53 for lokal DNS

# Installer en webtjener som vil begynne å lytte på port 80:
sudo apt install nginx # velg OK hvis du får spørsmål
                       # (bruk TAB- og ENTER-tastene)

# Er det nå et program som lytter på port 80?
ss -tlpn # kolonnen "Local Address:Port" skal liste port 80

# Fra hvilken katalog serverer nginx nettsider?
grep -r root /etc/nginx/ | grep www
# Det er her du kan legge nettsidene dine for å se dem
# på din lokale ip-adresse
```

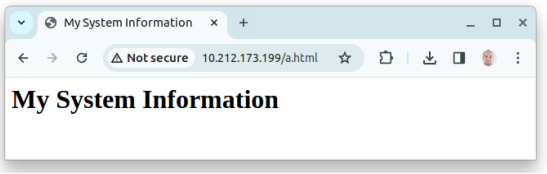
2. Hva er fordelen med å bruke et *here script* (aka *here document*) i stedet for bare `echo`-kommandoen med doble anførselstegn (eller enkle anførselstegn)?
3. Lag nettsiden ved hjelp av det siste eksempelskriptet du ser nederst på Here Scripts. Du skal:
   (a) Kopiere og lime inn skriptet i din egen fil.
   (b) Endre rettighetene på fila, slik at du kan kjøre den som et skript.

[2] CISK-studenter må velge det andre alternativet: installere en webtjener på din egen Linux-VM.

(c) Kjøre skriptet og omdirigere outputen til en HTML-fil.

(d) Kopiere HTML-fila til katalogen der nettsidene ligger (merk: hvis du er på Linux-VM-en må du sette `sudo` foran kommandoen, siden en vanlig bruker ikke har lov til å skrive til nginx sin rotkatalog).

(e) Vise fila i nettleseren din:



4. Hva er den spesielle egenskapen i dette skriptet (det nederst på Here Scripts) som ikke var en del av det forrige skriptet som også brukte et here script?

3.4 Variables

Les teksten og studer/gjør eksemplene på Variables. Tilleggsnotater fra meg:

- Husk at når du tilordner verdier til variabler, kan du ikke ha mellomrom rundt =, siden shellet da vil tolke det som en kommando med opsjoner i stedet for en tilordning.

Oppgaver:

1. Endre skriptet ditt til å inkludere en variabel og en miljøvariabel[3] (på samme måte som i Variables). Utfør de samme operasjonene som i forrige oppgave for å oppdatere nettsiden slik at den ser omtrent slik ut:

3.5 Command Substitution and Constants

Les teksten og studer/gjør eksemplene på Command Substitution and Constants. Tilleggsnotater fra meg:

- Kommandosubstitusjon er så kraftig! Den lar deg bruke outputen fra andre kommandolinjer eller skript i din nåværende kommandolinje (eller skript).

[3] Egentlig er `HOSTNAME` ikke en miljøvariabel i streng forstand, den lages automatisk av Bash, men du vil ikke se den når du kjører `printenv`.

Oppgaver:

1. Endre skriptet ditt til å inkludere kommandosubstitusjon i henhold til Command Substitution and Constants. Utfør de samme operasjonene som i forrige oppgave for å oppdatere nettsiden slik at den ser omtrent slik ut:

3.6 Flow Control - Part 1

Les teksten og studer/gjør eksemplene på Flow Control - Part 1. Tilleggsnotater fra meg:

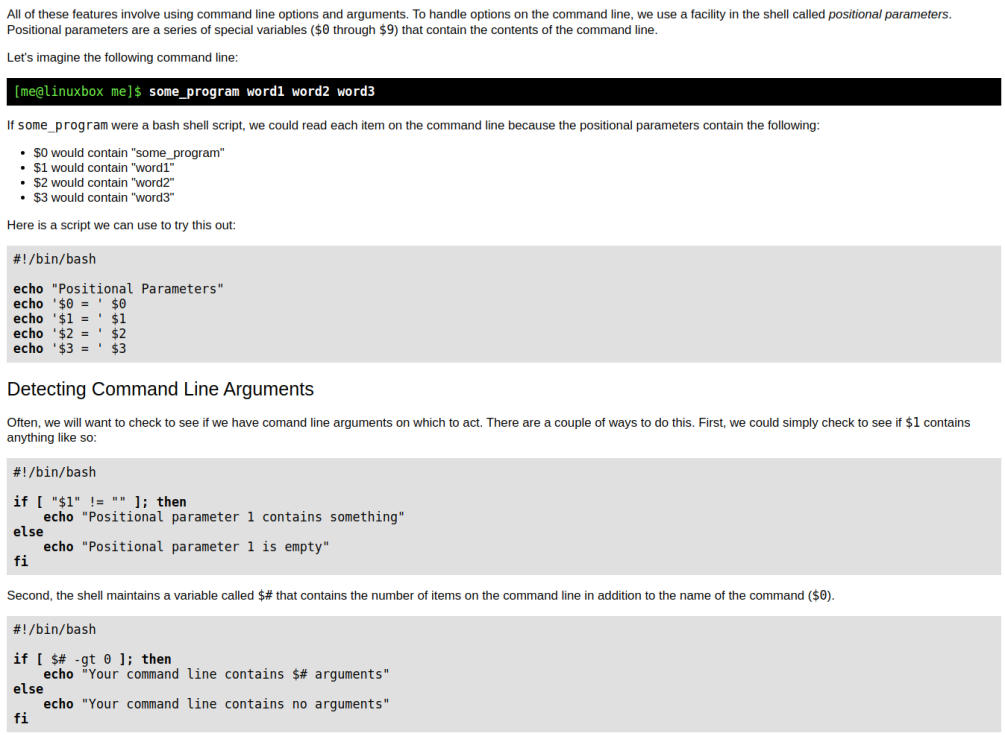
- The if command is fairly simple on the surface; it makes a decision based on the *exit status* of a command."

Oppgaver:

1. Kjør `help -m test | less` og søk i denne teksten etter `Arithmetic` så du forstår hvordan du kan sammenligne tall i if-tester.
2. Skriv en if-test som sjekker om -1 er mindre enn 0. Hvis sann skal den skrive ut "Yes, it's true!".
3. Hva er forskjellen på de to sammenligningsoperatorene = og `-eq`?
4. Lagre filnavnet `mysil` i en variabel (fila trenger ikke å eksistere). Skriv en if-else-test på kommandolinjen som sjekker om fila eksisterer. Hvis fila eksisterer, skal kommandoen skrive ut
   `mysil er her :)`
   Hvis fila ikke eksisterer, skal kommandoen skrive ut
   `mysil er ikke her :(`
5. Skriv et skript der den første kommandolinjen er:
   `read -p "Enter a string: " mysil`
   (dette leser en streng fra brukeren inn i variabelen `mysil`)
   Skriptet skal så sjekke om brukeren skrev inn en tom streng eller ikke.

3.7 Positional Parameters

Fra Positional Parameters trenger du bare å lese denne delen om *posisjonelle parametre* (å gi kommandolinjeargumenter til skriptet ditt):



All of these features involve using command line options and arguments. To handle options on the command line, we use a facility in the shell called *positional parameters*. Positional parameters are a series of special variables ($0 through $9) that contain the contents of the command line.

Let's imagine the following command line:

```
[me@linuxbox me]$ some_program word1 word2 word3
```

If `some_program` were a bash shell script, we could read each item on the command line because the positional parameters contain the following:

- $0 would contain "some_program"
- $1 would contain "word1"
- $2 would contain "word2"
- $3 would contain "word3"

Here is a script we can use to try this out:

```
#!/bin/bash

echo "Positional Parameters"
echo '$0 = ' $0
echo '$1 = ' $1
echo '$2 = ' $2
echo '$3 = ' $3
```

Detecting Command Line Arguments

Often, we will want to check to see if we have comand line arguments on which to act. There are a couple of ways to do this. First, we could simply check to see if $1 contains anything like so:

```
#!/bin/bash

if [ "$1" != "" ]; then
    echo "Positional parameter 1 contains something"
else
    echo "Positional parameter 1 is empty"
fi
```

Second, the shell maintains a variable called $# that contains the number of items on the command line in addition to the name of the command ($0).

```
#!/bin/bash

if [ $# -gt 0 ]; then
    echo "Your command line contains $# arguments"
else
    echo "Your command line contains no arguments"
fi
```

3.8 Flow Control - Part 3

Les teksten og studer/gjør eksemplene på Flow Control - Part 3. Tilleggsnotater fra meg:

- Husk alltid at du kan skrive en for-løkke direkte på kommandolinjen, du trenger ikke å legge den inne i et shell-skript.

Oppgaver:

1. Kjør `echo {01..05}` for å se at det genererer en liste med tall. Skriv en for-løkke som skriver ut disse tallene ett på hver linje.
2. Kjør `echo {a..c}` for å se at det genererer en liste med tegn. Skriv en nøstet for-løkke som skriver ut følgende:

```
01.a
01.b
01.c
02.a
02.b
02.c
03.a
03.b
03.c
```

3. Skriv et shell-skript som sjekker om det er nøyaktig tre kommandolinjeargumenter (det skal avslutte med feil hvis ikke tre), og behandler disse tre i en for-løkke for å echo-e dem slik:

```
$ ./argcheck.sh en to
$ ./argcheck.sh en to tre
en is an argument
to is an argument
tre is an argument
```

4. Skriv et shell-skript som itererer over alle kommandolinjeargumenter og sjekker om de er filer:

```
$ ./filecheck.sh a mysil fil.txt
a is a file
fil.txt is a file
```

5. Skriv et shell-skript som tar en liste med kataloger som kommandolinjeargumenter, teller antall filer og kataloger i hver av disse katalogene og skriver dem ut i sortert rekkefølge (det høyeste tallet øverst):

```
$ ./numfiles.sh /tmp /usr/share/ /usr/ /var
109 files and dirs in /usr/share/
14 files and dirs in /var
12 files and dirs in /usr/
7 files and dirs in /tmp
```

6. En operatør trenger ofte å sjekke om en liste med tjenere svarer. Skriv et skript som tar et vilkårlig antall verts-/domenenavn som kommandolinjeargumenter, og for hvert av dem skriver ut den første linjen av HTTP-svaret (statuslinjen). Bruk `curl -Is` der `-I` henter bare HTTP-hodet og `-s` demper framdriftsinfo, og `head -n 1`:

```
$ ./sjekk.sh nrk.no ntnu.no
HTTP/2 200
HTTP/2 200
```

7. (AVANSERT) Hvem eier IP-adressene i et nett? Verktøyet `whois` slår opp registreringsinformasjon for en IP-adresse (må installeres først: `sudo apt install whois`). Skriv en for-løkke *direkte på kommandolinjen* som for hver av adressene `1.1.1.1` til `5.1.1.1` slår opp `whois` og plukker ut linja med organisasjonsnavnet:

8. Vi inkluderer ikke shell-funksjoner som pensum, men legg merke til bruken av kommandoene `du`, `find` og `uname` i funksjonene nederst på Flow Control - Part 3.

Tabell 1: Relevante aliaser.

Linux	**PowerShell**
`pwd`	`Get-Location`
`ls`	`Get-ChildItem`
`cd`	`Set-Location`
`cp`	`Copy-Item`
`mv`	`Move-Item`
`rm`	`Remove-Item`
`cat`	`Get-Content`
`sort`	`Sort-Object`
`tee`	`Tee-Object`
`echo`	`Write-Output`
`ps`	`Get-Process`
`kill`	`Stop-Process`
`(wc)`	`Measure-Object`
`(head/tail)`	`Select-Object`

4 PowerShell

Basert på det vi har dekket i de første kapitlene om Linux, lister tabell 1 de relevante aliasene i PowerShell for noen av kommandoene du sannsynligvis allerede kjenner.

4.1 Fra C til PowerShell

I programmeringskurset ble du introdusert for datastrukturen som kalles en *struct* i programmeringsspråket C. En struct:

```
// Definer datatypen Tidspunkt:
struct Tidspunkt  {
  int time,
      minutt,
      sekund;
};
// Lag en instans (en variabel som heter tid)
// av datatypen Tidspunkt:
struct Tidspunkt tid;
// Tilordne verdier til medlemmene:
tid.time   = 17;
tid.minutt = 23;
tid.sekund = 57;
```

PowerShell er bygget rundt begrepet *objekter*, som ligner på struct-datastrukturene du kjenner fra C. PowerShell fungerer både som et programmeringsspråk og som et kommandolinjegrensesnitt. Selv om det ligner på Linux-kommandolinja, er den viktigste forskjellen at PowerShell opererer på objekter (også når du piper mellom dem) i stedet for enkle strømmer av bytes eller tegn, slik det er i Linux. Du vil møte PowerShell-objekter som representerer filer, kataloger, prosesser, brukere, grupper, pakker, logglinjer og mer. Kjører du for eksempel kommandoen `Get-Date`, får du tilbake et objekt av typen `DateTime`, og medlemmene i det — kalt *egenskaper* (properties) i PowerShell — kan aksesseres direkte:

```
PS> Get-Date
  Sunday, December 17, 2023 5:38:32 PM

PS> (Get-Date).DayOfWeek
  Sunday

PS> $tid = Get-Date

PS> $tid.DayOfWeek
  Sunday
```

Merk: Denne PowerShell-delen dekker de grunnleggende begrepene du trenger for å komme i gang, og den inneholder mer stoff enn vi rekker over på de fem timene vi har satt av. Les seksjonene i den rekkefølgen de står — de bygger på hverandre — og prioriter **GJØR DETTE**-punktene underveis og oppgavene til slutt. Alt du trenger her, kjører du lokalt på din egen Windows-maskin. Målet er ikke å pugge cmdleter, men å forstå den objektorienterte pipelinen, som er det som skiller PowerShell fra Linux-shellet.

Teksten inneholder lenker (uthevet i blått) til offisiell Microsoft-dokumentasjon for ulike PowerShell-begreper. Du behøver ikke lese alle, men de er der hvis du vil gå dypere inn i et tema.

4.2 Windows – introduksjon og bakgrunn

Windows har et GUI (Windows Shell), det gamle kommandolinjegrensesnittet `cmd` og det foretrukne kommandolinjegrensesnittet PowerShell (både `cmd` og PowerShell kan også kjøres i Windows Terminal).

Slik kopierer og limer du inn på kommandolinja:

- kopiere: marker teksten og trykk Enter
- lime inn: høyreklikk

4.3 Kom i gang med PowerShell

Merk: I framtiden går vi kanskje over til å bruke winget for å installere PowerShell Core og annen programvare. Foreløpig fortsetter vi imidlertid med Chocolatey slik det er beskrevet nedenfor

Windows leveres som standard med den eldre Windows PowerShell (versjon 5.1). I dette kurset bruker vi den nyere, plattformuavhengige versjonen som heter PowerShell Core. Merk at kjørbar fil for Windows PowerShell er `powershell.exe`, mens PowerShell Core bruker `pwsh.exe`. PowerShell Core kan også installeres på Linux og macOS, der det startes med den samme kommandoen `pwsh`. Slik installerer du PowerShell Core:

```
# søk på maskinen din etter powershell
# start PowerShell med "Run as administrator"
# kjør følgende kommandoer:
# (du kan kopiere og lime inn alle linjene på én gang)

Set-ExecutionPolicy Bypass -Scope Process -Force
[System.Net.ServicePointManager]::SecurityProtocol = ‘
  [System.Net.ServicePointManager]::SecurityProtocol -bor 3072
iex ((New-Object System.Net.WebClient).DownloadString(‘
  ’https://chocolatey.org/install.ps1’))
choco install -y powershell-core
exit
```

PowerShell bruker ExecutionPolicies som en sikkerhetsmekanisme for å hindre at skript kjøres ved et uhell, særlig skript som er lastet ned fra Internett. Hensikten er å beskytte deg mot å kjøre potensielt skadelig kode utilsiktet.

Noen ganger må du endre execution policy for at dine egne skript skal kunne kjøre.

- Bruk `Get-ExecutionPolicy` for å sjekke hvilken policy som gjelder på systemet ditt.
- Bruk `Set-ExecutionPolicy` for å endre den når det er nødvendig.

Execution Policy er ingen sterk sikkerhetsgrense. Det er en svært svak beskyttelse, siden enhver bruker med tilstrekkelige rettigheter enkelt kan endre policyen med `Set-ExecutionPolicy`. Du bør derfor ikke stole på den som et primært sikkerhetstiltak.

En mye brukt innstilling er `RemoteSigned`, som krever at ethvert skript lastet ned fra Internett må være digitalt signert av en betrodd utgiver før det kan kjøres.

Standardinnstillinger:

- Windows-klientsystemer som kjører Windows PowerShell bruker `Restricted`, som blokkerer alle skript. Windows-klientsystemer som kjører PowerShell Core bruker
`RemoteSigned`.
- Windows-tjenere bruker `RemoteSigned`.
- På systemer som ikke er Windows (som Linux eller macOS), er standardpolicyen `Unrestricted`, det vil si at skript kan kjøre uten krav om signatur.

Hvis du laster ned et skript som ikke er digitalt signert, kan du likevel kjøre det ved å oppheve blokkeringen manuelt med `Unblock-File`.

Hva er en digital signatur? En digital signatur er en kryptografisk mekanisme som verifiserer autentisiteten og integriteten til en fil. Den sikrer at skriptet ble laget av en betrodd utgiver og ikke er endret etter at det ble signert. Dette beskytter mot manipulert eller ondsinnet kode.

Chocolatey er en kraftig pakkebehandler for Windows som gjør det enkelt å installere og oppdatere programvare. Det er likevel viktig å forstå risikoen — særlig når det gjelder *sikkerhet i forsyningskjeden* (supply chain security). Når du installerer programvare fra fellesskapets pakkearkiver, stoler du på at pakkene ikke er manipulert.

Før du tar Chocolatey i utstrakt bruk, bør du lese om Chocolateys Rigorous Moderation Process for Community Packages for å forstå hvordan pakkene gjennomgås og godkjennes før du gjør deg avhengig av dem.

Hva er sikkerhet i forsyningskjeden? Sikkerhet i forsyningskjeden (supply chain security) handler om å sørge for at programvare og avhengighetene dens kommer fra betrodde kilder og ikke er kompromittert underveis i distribusjonen. En ondsinnet aktør kan injisere skadelig kode i en pakke eller i avhengighetene dens, så det er kritisk å verifisere integritet og autentisitet når du bruker pakkearkiver drevet av et fellesskap. Det samme tillitsproblemet har du på Linux-siden: når du kjører `sudo apt install nginx`, stoler du på at pakkearkivet og de som vedlikeholder pakka, er til å stole på.

Du kan bruke Chocolatey til å installere det meste av programvaren du trenger (slik vi gjorde med PowerShell Core). Det gjør det også enkelt å holde styr på oppdateringer:

- Se etter utdaterte pakker med `choco outdated`
- Oppgrader installerte pakker med `choco upgrade all`

Viktig: ikke kjør PowerShell som Administrator med mindre det er nødvendig

Før vi går videre: forsikre deg om at du ikke kjører PowerShell som Administrator med mindre du får eksplisitt beskjed om det. Forhøyede rettigheter skal bare brukes til systemadministrasjonsoppgaver som å installere programvare, legge til brukere, stoppe kritiske tjenester og lignende.

Hvorfor er dette viktig? Mennesker gjør feil — det gjør ikke datamaskiner. Hvis du ved et uhell kjører en skadelig kommando med administratorrettigheter, kan konsekvensene bli alvorlige. Ved å jobbe som en vanlig bruker med begrensede rettigheter reduserer du risikoen for å gjøre skade på hele systemet betraktelig.

4.4 Cmdleter, parametere, aliaser og hjelp

Kommandoene i PowerShell kalles *cmdleter* (uttales "commandlets") og følger syntaksen `verb-noun`, for eksempel `Write-Output`. Cmdleter brukes ofte sammen med *parametere* (merk: i Linux sine man-sider kalles parametere for *options*), og parametere kan ha verdier (også kalt *argumenter*).

Det anbefales å lese dokumentasjonen About command syntax for en dypere forståelse.

Lange kommandolinjer Noen ganger blir PowerShell-kommandolinjer for lange til å få plass på én linje. På samme måte som Linux har backslash (\) som tegn for linjefortsettelse, bruker PowerShell et tegn som kalles *backtick* (`).

Backtick er ikke nødvendig hvis linja slutter med et pipe-tegn (|). Når du ser en linje som slutter med backtick eller pipe, betyr det at kommandoen fortsetter på neste linje.

I begynnelsen kan det være vanskelig å huske alle de forskjellige cmdletene. Heldigvis har de fleste cmdletene aliaser som svarer til kommandoer du kanskje kjenner fra DOS (cmd.exe) eller Unix/Linux. I tillegg har PowerShell korte aliaser for mange cmdleter, slik at du slipper å skrive så mye. For å finne cmdleten som hører til en kommando du kjenner, kan du bruke cmdleten `Get-Alias`:

```
# finnes det en cmdlet som svarer til Unix/Linux ls?
Get-Alias ls
# er det mange aliaser til Get-ChildItem?
Get-Alias -Definition Get-ChildItem
# list alle aliaser
Get-Alias
# eller list dem fra "alias-disken" (tilsvarende C:\-disken din)
Get-ChildItem Alias:\
```

Aliaser er praktiske når du skriver raskt, men det anbefales å bruke de fullstendige cmdlet-navnene, særlig i skriptene dine. Fulle navn gjør skriptene mer lesbare, enklere å vedlikeholde og lettere for andre å forstå.

Hver cmdlet har flere parametere som er spesifikke for det den gjør, men det finnes også et sett med felles parametere som er tilgjengelige for de fleste cmdleter. Noen særlig nyttige fellesparametere er `-Verbose`, `-ErrorAction SilentlyContinue` og `-WhatIf`.

> **GJØR DETTE**: Prøv `Get-Service | Stop-Service -WhatIf`.
> Prøv `Get-ChildItem -Recurse C:\Windows\System32\wbem\MO*` med og uten opsjonen `-ErrorAction SilentlyContinue`.

Et program som kjører i Windows kalles en *prosess*, og de fleste prosesser inneholder én eller flere *tråder* som utfører instruksjoner samtidig. En tjeneste er en spesiell type prosess som kjører i bakgrunnen, ofte uten at noen bruker er logget inn. Tjenester tar seg typisk av oppgaver på systemnivå, som nettverk, utskrift eller sikkerhet. Noen ganger kan en prosess eller en gruppe prosesser pakkes inn i en job. Jobs lar kommandoer eller skript kjøre asynkront i bakgrunnen, men dette er mindre vanlig enn tjenester.

For å få hjelp med cmdleter bruker du cmdleten `Get-Help`. For eksempel:

- `Get-Help Write-Output | more` viser hjelpeteksten én side om gangen.

En nyttig mulighet er å se hjelpesiden på nett ved å legge til parameteren `-Online`:

- `Get-Help Write-Output -Online` åpner dokumentasjonen i nettleseren din.

Foretrekker du hjelp offline, kan du bruke:

- `Get-Help Write-Output -Examples` for å se praktiske eksempler.
- `Get-Help Write-Output -Full` for hele hjelpeteksten.

PowerShell støtter *TAB-fullføring* både for cmdleter og parametere. Da kan du skrive deler av en kommando eller en parameter og trykke TAB-tasten for å fullføre den automatisk. TAB-fullføring sparer tid, reduserer skrivefeil og hjelper deg å oppdage tilgjengelige kommandoer og opsjoner.

> **GJØR DETTE**: Bruk `Get-Help Select-String -online` (merk: `Select-String` ligner på Linux sin `grep`) og finn eksempelkoden for Find a string in subdirectories". Prøv den. Fikk du en tilgangsfeil (rød tekst)? Gjenta i så fall kommandoen, men legg til parameteren `-ErrorAction SilentlyContinue` til cmdleten `Get-ChildItem`. Det undertrykker feilmeldingene og lar kommandoen fortsette. Å søke gjennom filsystemer kan gå tregt. Hvor tregt? Gjenta kommandoen, men pakk den inn i krøllparenteser (`{ }`) og sett cmdleten `Measure-Command` foran. Da måler du hvor lang tid kommandoen bruker.

For å finne ut hvilke cmdleter du kan bruke, er hovedverktøyet cmdleten `Get-Command`, f.eks. Prøv `Get-Command *DNS*`. (Og litt på gøy: prøv `Get-Command | Get-Random | Get-Help`. Gjenta dette noen ganger for å utforske tilfeldige cmdleter og lære hva de gjør!)

4.5 Variabel, array og hashtable

En variabel i PowerShell kan representere hva som helst av objekter. Variabler kan også samles i samlinger som *arrayer* eller *hashtabeller*:

Array: En samling der hvert element aksesseres med en numerisk indeks (f.eks. `$myArray[0]`).

Hashtable: En samling av nøkkel-verdi-par der hvert element aksesseres med en nøkkel i stedet for en numerisk indeks (f.eks. `$myHash["Name"]`).

```
# En variabel kan inneholde hvilket som helst objekt
$a = 'hei'
$a.GetType().FullName          # System.String (strengobjekt)

# Bruk en filsti vi vet finnes, som demonstrasjon
# $PSHOME peker til installasjonskatalogen til PowerShell
$a = Get-Item "$PSHOME\powershell.exe"
$a.GetType().FullName          # System.IO.FileInfo (filobjekt)

# Arrayer: numerisk indeks som starter på 0
$a = @('frodeh','kelly')
$a.GetType().FullName          # System.Object[] (array)
$a[0]                          # 'frodeh'
$a[1]                          # 'kelly'
$a                             # Skriver ut de to elementene

# Legg til elementer i en array (lager en ny array bak kulissene)
$a += 'erik'
$a                             # Inneholder nå 'frodeh','kelly','erik'
$a.Count                       # 3

# Hashtabeller: nøkkel-verdi-par, indeksert med nøkkel (ikke tall)
$a = @{'frodeh'='Frode Haug'; 'kelly'='Jia-Chun Lin'}
$a.GetType().FullName          # System.Collections.Hashtable (hashtable)
$a['kelly']                    # 'Jia-Chun Lin'
$a[1]                          # Vanligvis $null - hashtabeller bruker
                               # nøkler, ikke numerisk indeks
$a                             # Skriver ut nøkkel-verdi-parene
$a.Count                       # Antall oppføringer (2)
$a.Keys                        # Lister nøklene: 'frodeh','kelly'

# Legg til eller oppdater oppføringer i en hashtable
$a['erik'] = 'Erik Hjelmås'    # Legg til nytt nøkkel-verdi-par
$a['kelly'] = 'Kelly Lin'      # Oppdater verdien til en eksisterende nøkkel
```

```
$a.Keys                          # 'frodeh','kelly','erik'
```

> **GJØR DETTE**: 1. Bruk cmdleten `Write-Output` og hashtabellen `$a` over (det siste eksempelet) og skriv en kommandolinje som skriver ut "User frodeh has real name Frode Haug". 2. Gjenta oppgaven, men bruk denne gangen formatoperatoren `-f` (se Format operator -f). (hint: i den første oppgaven må du bruke `$(...)` for å sette verdien fra hashtabellen inn i strengen, mens du i den andre oppgaven ikke trenger dette.)

4.6 Objekter og `Get-Member`

Poweren i PowerShell kommer av at du jobber med *objekter* og ikke bare ren tekst eller strømmer av bytes. Det betyr at når du piper data mellom cmdleter, sender du rike objekter med egenskaper og metoder — ikke bare strenger.

Eksempel: å slå opp et DNS-navn Vil du finne IP-adressen til et fullstendig domenenavn (FQDN — Fully Qualified Domain Name), som `ftp.uninett.no`, kan du bruke det gammeldagse programmet `nslookup` eller den moderne PowerShell-cmdleten `Resolve-DnsName`:

```
# Med nslookup (gammelt verktøy)
nslookup.exe ftp.uninett.no
nslookup.exe ftp.uninett.no | Get-Member
# Legg merke til: TypeName er System.String - nslookup gir bare ren tekst.

# Med Resolve-DnsName (PowerShell-cmdlet)
Resolve-DnsName ftp.uninett.no
Resolve-DnsName ftp.uninett.no | Get-Member
# Legg merke til: TypeName er Microsoft.DnsClient.Commands.DnsResponseRecord
# Dette er et objekt med egenskaper som IP4Address.
```

Å aksessere egenskaper Siden `Resolve-DnsName` returnerer et objekt, kan du enkelt aksessere egenskapene:

```
(Resolve-DnsName ftp.uninett.no).IP4Address
```

Å gjøre det samme med `nslookup` krever komplisert tekstparsing:

```
(nslookup.exe ftp.uninett.no 2>&1 | ‘
  Select-String '\d+\.\d+\.\d+\.\d+').Matches[-1].Value
```

Hvorfor `Get-Member`? Når du piper objekter, bruker du `Get-Member` for å undersøke hvilke egenskaper og metoder objektet inneholder. For å aksessere en egenskap eller en metode bruker du:

```
Objekt.Egenskap
(Cmdlet-som-returnerer-objekt).Egenskap
```

> **GJØR DETTE**: Gjør `$name = 'mysil'`. Bruk egenskapene og metodene til `$name`-objektet til å
>
> - finne ut hvor mange tegn strengen inneholder.
> - skrive ut strengen med store bokstaver.

4.7 Select-Object

Cmdleten `Select-Object` er nyttig i flere sammenhenger:

1. Å undersøke egenskaper Du kan bruke `Select-Object` til å vise verdiene til alle egenskapene til et objekt. Det er iblant mer praktisk enn `Get-Member` når du skal bli kjent med egenskapene til et objekt:

```
Resolve-DnsName ftp.uninett.no | Select-Object -Property *
```

2. Å lage et redusert objekt Du kan lage et nytt objekt i pipelinen med et redusert sett med egenskaper. Sammenlign outputen fra disse to kommandoene:

```
# Lag en fil
Write-Output mysil > abc.txt

# Fullt objekt med alle egenskaper
Get-ChildItem .\abc.txt | Get-Member

# Redusert objekt med utvalgte egenskaper
Get-ChildItem .\abc.txt | ‘
  Select-Object -Property Name, LastWriteTime | Get-Member
```

3. Å oppføre seg som head/tail i Unix/Linux `Select-Object` kan også brukes til å velge de første eller de siste elementene i en samling, tilsvarende `head` og `tail` i Unix/Linux:

```
Get-Process | Select-Object -Last 5
Get-ChildItem | Select-Object -First 3
```

> **GJØR DETTE:** Kjør `Get-Content C:\Windows\System32\drivers\etc\services` og pipe det til `Select-Object` slik at du bare skriver ut linje 30 til 50 (altså: fra linjenummer 30 til linjenummer 50).

4. Avansert: å lage nye egenskaper underveis Iblant bruker vi `Select-Object` til å lage nye egenskaper dynamisk. Vil vi for eksempel vise eieren av filer (som ikke finnes direkte som en egenskap), kan vi bruke en beregnet egenskap med en hashtable:

```
Get-ChildItem |
  Select-Object Name, Directory, @{Name="Owner"; ‘
    Expression={(Get-ACL $_.FullName).Owner}}, ‘
    CreationTime, LastAccessTime | Format-Table
```

Legg merke til syntaksen for beregnede egenskaper:

- **Name**: overskriften.
- **Expression**: en skriptblokk som beregner verdien for hvert objekt i pipelinen (`$_`).

4.8 Filtrering med `Where-Object`

Bruk den automatiske variabelen `$_` for å referere til det gjeldende objektet i pipelinen. `Where-Object` filtrerer objekter på samme måte som `grep` i Unix/Linux.

```
# List alle tjenester:
Get-Service

# Tjenester der Status er nøyaktig lik ’Running’:
Get-Service | Where-Object { $_.Status -eq ’Running’ }

# Kombiner betingelser (-eq for likhet, -like for jokertegn):
Get-Service | Where-Object { $_.Status -eq ’Running’ -and ‘
  $_.StartupType -like ’*auto*’ }

# Bruk -match for regulære uttrykk:
Get-Service | Where-Object { $_.Status -eq ’Running’ -and ‘
  $_.StartupType -match ’.*auto.*’ }
```

Bla gjennom dokumentasjonen for alle sammenligningsoperatorene: About Comparison Operators.

> **GJØR DETTE:** Start enda en instans av PowerShell Core slik at du har minst to som kjører. Bruk `Get-Process` og `Where-Object`:
>
> - List alle prosesser der `Name` er lik `pwsh`.
> - List alle prosesser der `WorkingSet` er større enn 10MB.
> - Bruk `Get-Member` eller `Select-Object -Property *` for å finne egenskapen som viser hele stien til `pwsh.exe`.

Tips: For effektivitetens skyld bør du filtrere så langt til venstre som mulig:
`Get-Process -Name notepad` er bedre enn
`Get-Process | Where-Object { $_.Name -eq 'notepad' }`.

4.9 Formaterings- og output-cmdleter: `Format-*` og `Out-*`

Disse cmdletene brukes til å *formatere eller omdirigere den endelige outputen* fra en pipeline. **Viktig regel:** *filtrer til venstre, formater til høyre*. Gjør filtrering og utvalg før formatering.

Format-*-cmdleter Eksempler: `Format-Table`, `Format-List`. Hensikt: styre hvordan data vises på skjermen. *Bruk dem bare til visning*, ikke til å lagre data eller til videre bearbeiding. Skal du velge ut egenskaper for lagring eller videre bearbeiding, bruker du `Select-Object` i stedet.

```
Get-ChildItem -File C:\Windows\ |
  Format-List -Property Name, Length, LastAccessTime
Get-ChildItem -File C:\Windows\ |
  Format-Table -Property Name, Length, LastAccessTime
```

Out-*-cmdleter Eksempler: `Out-File`, `Out-GridView`, `Out-Printer`. Hensikt: sende output til et annet sted enn konsollet.

Vanlig bruk:

```
# Lagre til en fil
Get-ChildItem -File b* | Out-File b.dat    # Det samme som > b.dat

# Send til en grafisk tabellvisning
Get-ChildItem -File b* | Out-GridView
```

```
# Kopier til utklippstavla
Get-ChildItem -File b* | clip

# Eksporter til CSV
Get-ChildItem -File b* | Export-Csv $home\a.csv
```

> **GJØR DETTE:** TAB deg gjennom alle `ConvertTo-`-cmdletene for å se hvilke muligheter du har til å konvertere objekter til ulike formater.

Tips: PowerShell legger automatisk til `Out-Default` på slutten av enhver pipeline, noe som iblant kan gi uventet oppførsel. Se eksempelet med bruk av semikolon (det samme kan skje inne i skript/funksjoner).

4.10 Sortering av objekter med `Sort-Object`

Cmdleten `Sort-Object` lar deg sortere objekter etter hvilken som helst egenskap. Det er svært nyttig når du jobber med store datamengder, eller når du trenger å organisere output for enklere analyse.

Nedenfor er to praktiske eksempler, der `Get-Date` brukes til å filtrere ut filer som nylig har vært aksessert. Merk at PowerShell-kommandoer kan gå over flere linjer:

```
# Sorter filer aksessert de siste 30 minuttene etter størrelse (synkende)
Get-ChildItem C:\Windows\System32\ |
  Where-Object { $_.LastAccessTime -gt (Get-Date).AddMinutes(-30) } |
    Sort-Object -Property Length -Descending

# Samme filter, men velg ut de 10 minste filene
Get-ChildItem C:\Windows\System32\ |
  Where-Object { $_.LastAccessTime -gt (Get-Date).AddMinutes(-30) } |
    Sort-Object -Property Length |
      Select-Object -Last 10
```

Hvorfor tidsstempler er viktige

Tidsstempler er kritiske i mange sammenhenger:

- Logganalyse og etterforskning (å bygge tidslinjer).
- Sikkerhetskopiering og gjenoppretting.

- Hverdagslige oppgaver, som å finne ut hvilken fil du redigerte eller lastet ned sist torsdag.

  GJØR DETTE: List alle prosesser der navnet matcher jokeruttrykket `'*host*'` (altså at prosessnavnet inneholder ordet `host`). Sorter outputen etter egenskapen `CPU`, og pipe deretter til `Format-Table` og vis bare egenskapene `Name`, `Id` og `CPU`.

4.11 Måling av objekter med `Measure-Object`

Du har allerede brukt `Measure-Command` til å måle kjøretid. Vil du telle objekter, linjer, ord eller tegn, eller regne ut statistikk fra egenskapene til objekter, bruker du `Measure-Object`.

```
# Tell hvor mange objekter som ble returnert i DNS-eksempelet
Resolve-DnsName ftp.uninett.no | Measure-Object

# Tell linjer, ord og tegn i en tekstfil
Get-Content C:\Windows\System32\drivers\etc\services |
  Measure-Object -Line -Word -Character

# Tell filer og regn ut total størrelse i System32
Get-ChildItem -File C:\Windows\System32\ | Measure-Object

# Regn ut sum og gjennomsnitt av filstørrelsene
Get-ChildItem -File C:\Windows\System32\ |
  Measure-Object -Property Length -Sum -Average
```

Denne cmdleten er nyttig til:

- å telle objekter i en pipeline.
- å regne ut statistikk som sum, gjennomsnitt, minimum og maksimum.
- å analysere tekstfiler med hensyn på linjer, ord og tegn.

  GJØR DETTE: Gjenta forrige oppgave (prosesser med navn `*host*`), og bruk `Measure-Object` til å:

  - telle antall prosesser.
  - regne ut gjennomsnittet av egenskapen `WorkingSet` (minnebruk).
  - lagre outputen i en variabel `$avghostproc`.
  - skrive ut gjennomsnittsverdien i megabyte (se eksempelet i seksjonen om hashtabeller).
  - formatere verdien med to desimaler ved hjelp av: `{0:N2} -f (uttrykk)` (se også Format operator -f).

4.12 Iterasjon med `ForEach-Object`

Mens `Where-Object` brukes til å filtrere objekter i pipelinen, brukes `ForEach-Object` når du vil *utføre en handling på hvert objekt* i pipelinen. Det gir deg fleksibilitet til å gjøre dine egne operasjoner.

Du kan også bruke vanlige løkker og betingelser som `foreach` eller `if`, som typisk brukes i skript (samlinger av kommandoer lagret i en fil).

```
# Enkel foreach-løkke: åpne Notepad 4 ganger
foreach ($i in 1..4) { notepad }

# Terminer alle Notepad-prosesser med en egen melding
Get-Process notepad | ForEach-Object {
    Write-Output "Terminating Notepad with id $($_.Id)"
    $_.Kill()
}
```

Merk: Vil du bare stoppe alle Notepad-prosessene, kan du ganske enkelt bruke:

```
Get-Process notepad | Stop-Process
```

> **GJØR DETTE:** Start en pipeline med strengene
> `'Set-Location', 'Get-Location', 'Push-Location'`
> og bruk `ForEach-Object` til å iterere over disse cmdlet-navnene. For hvert navn kaller du: `Get-Alias -Definition $_`

4.13 Betinget logikk: `if`, sammenligninger og forgrening

PowerShell støtter betinget logikk med `if`-setningen. Da kan du kjøre kode bare når en betingelse evaluerer til `True`. Betingelser involverer ofte sammenligningsoperatorer som `-eq` (lik) og `-gt` (større enn), og logiske operatorer som `-and` eller `-or`.

```
# Sjekk om Administrator-kontoen er aktivert
if ( (Get-LocalUser -Name Administrator).Enabled ) {
    Write-Output "It's enabled!"
}

# Sjekk om en streng ikke er tom
if ( $name.Length -gt 0 ) {
    Write-Output "Name has characters"
}
```

```
# Sjekk om en fil finnes
if ( Test-Path C:\Windows\System32\ntoskrnl.exe ) {
    Write-Output "OS exists!"
}
```

Viktige poeng å huske:

- Bruk parenteser `(...)` rundt betingelsen.
- Omslutt kodeblokken med krøllparenteser `{ ... }`.
- Kombiner betingelser med logiske operatorer (`-and`, `-or`).
- Utforsk sammenligningsoperatorene i About Comparison Operators.

Iblant er det enklere å bruke `Compare-Object`, for eksempel når du bare skal kopiere nye eller oppdaterte filer mellom to kataloger.

```
# Definer kilde- og målkatalog
$src = Get-ChildItem -Recurse "C:\Users\Admin\test\"
$dst = Get-ChildItem -Recurse "C:\Backup\"

# Håndter tomme samlinger for å unngå feil
if ($src -eq $null -or $dst -eq $null) {
    Write-Output "Compare-Object does not like empty objects :)"
    return
}

# Sammenlign på LastWriteTime og kopier de nyeste filene
Compare-Object -ReferenceObject $dst -DifferenceObject $src `
    -Property LastWriteTime -PassThru |
        Where-Object { $_.SideIndicator -eq '=>' } |
            Copy-Item -Recurse -Force -Destination C:\Backup\
```

Når du bruker `Compare-Object` med opsjonen `-PassThru`, legger PowerShell til en egenskap som heter `SideIndicator` på hvert objekt i pipelinen. Denne egenskapen forteller hva forskjellen er:

- `=>` betyr at objektet bare finnes i `DifferenceObject` (kilden).
- `<=` betyr at objektet bare finnes i `ReferenceObject` (målet).

Du kan bruke `SideIndicator` til å filtrere og velge ut objekter ut fra hvordan de er forskjellige.

4.14 Utvide PowerShell med moduler

PowerShell kan utvides med moduler som gir ny funksjonalitet. Vil du for eksempel styre Windows Update fra PowerShell, kan du installere modulen `PSWindowsUpdate`.

```
# Finn tilgjengelige moduler knyttet til Windows Update
Find-Module *windowsupdate*

# Installer modulen PSWindowsUpdate (kjør som Administrator)
Install-Module -Name PSWindowsUpdate

# Se etter oppdateringer
Get-WindowsUpdate

# Last ned oppdateringer
Get-WindowsUpdate -Download

# Installer oppdateringer (valgfritt, kan ta tid)
# Get-WindowsUpdate -Install
```

4.15 Lage og bruke PowerShell-skript

Et PowerShell-skript er ganske enkelt en samling kommandoer lagret i en fil (vanligvis med etternavnet `.ps1`). Det finnes flere måter å lage en skriptfil på:

- Bruk en teksteditor (f.eks. VS Code, micro, nano, vim, Notepad).
- Bruk `Out-File` (`>`) eller legg til på slutten med `Out-File -Append` (`>>`).
- Bruk et here document (kalt en here-string i PowerShell).

For små skript er en here-string praktisk. Eksempelet nedenfor lager et skript `myscript.ps1` som tar imot en parameter og viser størrelsen på fila:

```
@'
param($Name)
"Hello $Name, the size of this file is in Bytes:"
(Get-ChildItem .\myscript.ps1).Length
'@ > myscript.ps1
```

GJØR DETTE: Lag skriptet over og kjør det:

- Med og uten at du oppgir parameteren `Name`.

- Med og uten at du gir `Name` en verdi.

Iblant vil du se `&` som prefiks når skript eller kommandoer kjøres, særlig når du bruker TAB-fullføring. Døp for eksempel om `myscript.ps1` til `my script.ps1` og prøv TAB-fullføring. `&` er *kalloperatoren* (the call operator). Les mer i About Operators.

Å sjekke kvaliteten på skript Sjekk alltid kodekvaliteten i skriptene dine. PowerShell har verktøy for dette:

- PSScriptAnalyzer: en statisk kodeanalysator for PowerShell.
- Pester: et testrammeverk for større prosjekter.

```
# Installer PSScriptAnalyzer
Install-Module -Name PSScriptAnalyzer

# Analyser kvaliteten på skriptet
Invoke-ScriptAnalyzer myscript.ps1
# Ingen output betyr at ingenting ble funnet :)
```

4.16 Drives, profiler, variabler og navnerom

PowerShell innfører begrepet *drives* (disker), som er datalagre du kan aksessere som et filsystem. Det kan virke uvant til å begynne med, men det er svært praktisk, siden du kan gjenbruke det samme settet med kommandoer for å aksessere ulike typer data.

Bruk `Get-PSDrive` for å liste alle tilgjengelige drives. Naviger (`cd`) inn i drives du ikke visste om, og utforsk innholdet med `Get-ChildItem` eller `Get-ChildItem -Recurse`. For eksempel:

```
Get-ChildItem -Path Cert:\LocalMachine\CA
```

Denne kommandoen lister sertifikatutstederne (Certificate Authorities) på den lokale maskinen.

Variabler i PowerShell lagres som standard i navnerommet `variable`. PowerShell støtter flere navnerom, og iblant må du oppgi dem eksplisitt.

```
# Å jobbe med variabler
$a = 'heisan'
Write-Output $a
Write-Output $variable:a
```

```
# Miljøvariabler
cd $env:homepath
$env:homepath
$homepath
$home
$env:home

# Hvor profil-skriptet ligger
$profile
Get-Content $profile   # Tilsvarer 'cat' i Linux
```

- *Miljøvariabler* lagres i navnerommet `env`. Les mer: About Environment Variables.
- En *profil* er et skript som kjøres når PowerShell starter. Det kan brukes til å tilpasse miljøet ditt (f.eks. aliaser, funksjoner, innstillinger). Les mer: About Profiles.

Oppgaver:

1. Skriv en PowerShell-kommandolinje som lister *bare kataloger* (ikke filer) i hjemmekatalogen din og i alle underkatalogene under den. Kommandoen skal altså søke *rekursivt* gjennom alle kataloger under hjemmekatalogen din.
   (hint: bruk `Get-Help -Online Get-ChildItem` for å se hvilke parametere som finnes.)
2. Del denne kommandolinja opp i fire linjer med backtick (`), slik at hver parameter står på sin egen linje.
   `Get-ChildItem -Recurse -Path C:\Windows\System32\wbem\MO* -File`
3. Bruk cmdleten `Get-Date` til å lagre dato og klokkeslett akkurat nå i en variabel som heter `$now`. Bruk deretter variabelen `$now` til å:
   (a) skrive ut bare året (Year).
   (b) skrive ut bare ukedagen (DayOfWeek).
   (c) skrive ut bare timen (Hour).
   (d) skrive ut bare klokkeslettet (TimeOfDay).
   (e) konvertere tidspunktet til universaltid (UTC).
   (f) skrive ut dato og klokkeslett for 45 minutter siden.
   (g) skrive ut dato og klokkeslett om 3 måneder.
4. Du skal lage en liste over tjenernavn til et vedlikeholdsskript. Lag en array som heter `$servers` og som inneholder disse tre tjenerne:
   `'web01','db01','filesrv01'`
   Deretter skal du:

(a) skrive ut den første tjeneren i lista.

(b) legge til en ny tjener `'backup01'` i arrayen.

(c) vise hvor mange tjenere det er i arrayen.

5. Du skal dokumentere de lokale gruppene på en Windows-tjener. Først fyller du en hashtable `$groups` (nøkkel = gruppenavn, verdi = gruppebeskrivelse) med disse to linjene:

```
$groups = @{}
Get-LocalGroup | ForEach-Object { $groups[$_.Name] = $_.Description }
```

Deretter skal du:

(a) skrive ut hele hashtabellen.

(b) skrive ut beskrivelsen til gruppa `'Administrators'`.

(c) legge til en ny oppføring som kobler `'HelpDesk'` til `'Tier-1 support team'`.

(d) endre beskrivelsen til `'Users'` til `'Built-in standard users (non-admins)'`.

(e) vise hvor mange oppføringer hashtabellen har.

6. Gjør det samme som i oppgave 3, men uten å bruke en mellomvariabel — altså direkte på outputen fra `Get-Date`:

(a) Skriv ut året (Year).

(b) Skriv ut ukedagen (DayOfWeek).

(c) Skriv ut dato og klokkeslett for 30 minutter siden.

(d) Skriv ut dato og klokkeslett om 2 måneder.

7. *Select-Object — fire bruksmåter (lokale grupper).*
Bruk `Get-LocalGroup` og `Select-Object` på fire måter:

(a) Undersøk alle egenskaper: vis alle egenskapene til én gruppe (f.eks. den første) med `-Property *`.

(b) Lag et redusert objekt: vis bare `Name`, `Description` og `SID` (Security Identifier) for alle gruppene.

(c) Head/tail: vis bare de fem siste gruppene.

(d) (AVANSERT) Beregnet egenskap: lag en kolonne `MemberCount` som teller hvor mange medlemmer hver gruppe har, og vis `Name`, `MemberCount` og `Description`.

8. Skriv en PowerShell-kommandolinje som lister alle prosesser som har egenskapen `StartTime` innenfor de siste 12 timene. (hint: bruk `Get-Date` sammen med `AddHours()`.)

9. Skriv en PowerShell-kommandolinje som lister alle filer i katalogen `C:\Windows` som er større enn 10 KB. Utvid deretter pipelinen slik at den:

   (a) sorterer outputen etter filstørrelse (`Length`) i synkende rekkefølge.

   (b) viser bare de tre største filene.

   (c) viser bare egenskapene `Name`, `Length`, `LastAccessTime` og `LastWriteTime`.

   (hint: bruk `Where-Object`, `Sort-Object` og `Select-Object`.)

10. Bruk `Get-Process` til å liste alle prosessene som kjører. Eksporter bare egenskapene `Name`, `Id` og `CPU` til en CSV-fil som heter `processes.csv` i hjemmekatalogen din. Kontroller til slutt eksporten ved å åpne fila i Notepad.
    (hint: bruk `Select-Object`, `Export-Csv` og `notepad`.)

11. Skriv en PowerShell-kommandolinje som lister *navnet på alle kataloger* (fra katalogen du står i) som inneholder *minst 10 elementer* (filer eller underkataloger). Merk: du må kanskje bytte til en katalog med mange underkataloger/filer for å teste kommandoen, f.eks.
    `cd 'C:\Program Files\WindowsPowerShell\'`
    (hint: bruk `Measure-Object` til å telle elementene i hver katalog.)

5 C programming

En fin video å se sammen med denne teksten er Writing a simple Program in C (men merk at den bruker vi/vim-editoren, som jeg anbefaler alle som er virkelig interessert i Linux å lære, men de av dere som vil ha en enklere vei videre bør bruke enten nano eller micro istedet).

5.1 Kompilering og kjøring

Et veldig enkelt C-program ser slik ut:

```
#include<stdio.h>

int main(void) {
    printf("Yo Mysil\n");
    return 0;
}
```

La oss forklare det linje for linje:

`#include<stdio.h>`
Dette forteller kompilatoren at koden som følger vil bruke funksjoner (i vårt tilfelle: printf-funksjonen) som er definert i biblioteksfila `stdio.h`

`int main(void) {`
Programmet begynner å kjøre ved starten av funksjonen som heter `main`. `int` betyr at programmet skal gi en *retur-/exit-kode/-verdi/-status* (noen ganger sier vi exit status", noen ganger sier vi returverdi", osv.) og at den skal være et heltall. `void` er en måte å si at denne funksjonen ikke tar noen argumenter (eller parametre"om du vil). Til slutt indikerer den åpnende krøllparentesen at nå kommer selve koden".

`printf("Yo Mysil\n");`
`printf` er en ekstremt nyttig og mye brukt funksjon på tvers av mange programmeringsspråk (den finnes også på kommandolinjen i Linux). Argumentet til printf er i dette tilfellet teksten `Yo Mysil`, som vil bli skrevet ut på skjermen med et påfølgende linjeskift indikert av `\n`. Programsetningen avsluttes med et semikolon `;`.

`return 0;`
Dette er exit-statusen nevnt over. Programmer bør returnere en exit- status på `0` hvis de fullfører vellykket. Hvis et program ikke avslutter vellykket, f.eks. hvis det mangler input, vil det typisk ha en if-setning som sjekker en betingelse og returnerer `1` i stedet for `0` (eller noen andre verdier avhengig av typen feil).

```
}
```
Den lukkende krøllparentesen indikerer slutten på main-funksjonen, og dermed slutten på programmet.

Hvis vi lager dette programmet, kalt `mysil.c`, med et tekstredigeringsprogram som `nano` eller `micro`, eller ved å bruke et here document, kan vi bekrefte innholdet med `cat`:

```
$ cat mysil.c
#include<stdio.h>

int main(void) {
    printf("Yo Mysil\n");
    return 0;
}
```

For å kompilere og kjøre programmet gjør vi:

```
$ gcc -Wall mysil.c -o mysil
$ ./mysil
Yo Mysil
```

La oss forklare det linje for linje:

`$ gcc -Wall mysil.c -o mysil`
`gcc` er kompilatoren vår (GNU project C and C++ compiler). Når vi har en så fin kompilator, ber vi den hjelpe oss å skrive god kode ved å bruke kommandolinjeopsjonen `-Wall`, som er kort for enable all warningseller "Warnings all". `mysil.c` er kildekoden vi vil kompilere. Til slutt ber vi kompilatoren lage den kjørbare fila `mysil` fra kompileringsprosessen. Hvis vi ikke spesifiserer dette `-o mysil`, vil kompilatoren lage en kjørbar fil kalt `a.out` i stedet. Siden vi liker å identifisere hva programmet vårt er ut fra navnet, spesifiserer vi det typisk med `-o`-opsjonen, siden navnet `a.out` ikke forteller oss noe. Legg merke til logikken i kommandolinjeopsjonene:

- `gcc -Wall -o mysil mysil.c` er også riktig.
- `gcc -o mysil mysil.c -Wall` er også riktig.
- `gcc -o -Wall mysil mysil.c` er IKKE riktig.
- `gcc -Wall mysil -o mysil.c` er IKKE riktig.

`$ ./mysil`
Vi kjører programmet ved å sette `./` foran (med mindre programmet ligger i en katalog spesifisert av miljøvariabelen `PATH`). Vi trenger ikke å endre rettighetene på programmet slik vi måtte med shell- skript. Kompilatoren har tatt seg av dette for oss.

5.2 printf-funksjonen

La oss se på et annet C-program kalt `printfdemo.c`:

```
$ cat printfdemo.c
#include<stdio.h>
int main(void) {
    int x = 42;
    char str[] = "Mysil";
    printf("Tall er %d og tekst er %s\n",x,str);
    return 0;
}
```

La oss forklare de nye linjene linje for linje:

`int x = 42;`
Dette lager en variabel som heter `x` med en verdi `42`, og variabelen er av datatypen heltall (integer).

`char str[] = "Mysil";`
C har ikke sin egen datatype for en streng, så vi lager en streng som en array (tabell) av datatypen tegn (character). I dette tilfellet heter variabelen `str` med en "verdi"`Mysil` (vi sier "verdi"i anførselstegn siden det faktisk er en array av tegn og ikke en enkelt verdi).

`printf("Tall er %d og tekst er %s\n",x,str);`
printf-familien av funksjoner brukes til å formatere og skrive ut tekst. Vi vil i all hovedsak bare bruke `printf`-funksjonen, som skriver til skjermen, men du bør merke deg at det finnes en `fprintf` som vi kan bruke til å skrive til fil, og en `snprintf` som skriver til en strengvariabel (en char- array), samt flere andre lignende varianter av printf.
`"Tall er %d og tekst er %s\n"`
er en *format string* (formatstreng). En formatstreng inneholder *conversion specifiers* (konverteringsspesifikatorer), i vårt tilfelle er konverteringsspesifikatorene `%d` som sier erstatt meg med et heltall (et siffer)", og `%s` som sier erstatt meg med en streng". Konverteringsspesifikatorene erstattes med verdier fra argumentene som følger etter formatstrengen, og som standard erstatter de dem i den rekkefølgen de er listet:

- `%d` vil bli erstattet med verdien av `x` (42)
- `%s` vil bli erstattet med verdien av `str` (Mysil)

La oss kompilere og kjøre programmet for å bekrefte:

```
$ gcc -Wall printfdemo.c -o printfdemo
```

```
$ ./printfdemo
Tall er 42 og tekst er Mysil
```

La oss se på noen interessante detaljer ved formatstrenger ved hjelp av en forenklet versjon av `printfdemo.c` som bare bruker heltall:

Lese fra minnet Hva om vi gjør en feil og har flere konverteringsspesifikatorer enn vi har argumenter:

```
#include<stdio.h>
int main(void) {
    int x = 3, y = 5;
    printf("Tall er %d og %d og %d\n",x,y);
    return 0;
}
```

Vi kompilerer og kjører:

```
$ gcc -Wall printfdemo.c -o printfdemo
<vi får noen advarsler, men det kompilerer>
$ ./printfdemo
Tall er 3 og 5 og 288173504
```

HVA? Hvor kom `288173504` fra? Printf fortsetter å lese fra datamaskinens minne hvis vi ikke har spesifisert nok argumenter! Med andre ord: vi har en *memory leak* (minnelekkasje), og minnelekkasjer kan være ganske ille, se Heartbleed for en av de mest kjente minnelekkasje-sårbarhetene.

Skrive til minnet Printf har en spesiell konverteringsspesifikator `%n`:

```
#include<stdio.h>
int main(void) {
    int x = 3, y = 5, z;
    printf("Tall er %d og %d.%n\n",x,y,&z);
    printf("z er %d\n",z);
    return 0;
}
```

Vi kompilerer og kjører:

```
$ gcc -Wall printfdemo.c -o printfdemo
$ ./printfdemo
Tall er 3 og 5.
z er 15
```

HVA? Hvor kom 15 fra? Konverteringsspesifikatoren `%n` *skriver* antall tegn som er skrevet så langt inn i den tilhørende variabelen[4] (`z`) i argument- lista. 15 kommer av at tekststrengen Tall er 3 og 5."har totalt 15 tegn. *Vi kan skrive til en datamaskins minne ved hjelp av printf!*

Kontrollert skriving til minnet Konverteringsspesifikatorene i printf har også muligheten til å spesifisere hvilket argument i argumentlista de behandler, ved å legge til `m$` etter prosent-tegnet, der `m` er argumentnummeret, f.eks.
`printf("Tall er %2$d og %1$d\n",x,y);`
vil bytte om rekkefølgen på `x` og `y` i outputen, fordi `2$` i den første konverteringsspesifikatoren ber den sette inn det andre argumentet `y` i stedet for det første argumentet `x` (som er standardoppførselen). Dette betyr at hvis vi bruker denne opsjonen (`m$`) sammen med konverteringsspesifikatoren `%n`, kan vi kontrollere hvilken variabel vi skriver til:

```
#include<stdio.h>
int main(void) {
    int x = 3, y = 5;
    printf("Tall er %d og %d.%2$n\n",x,&y);
    printf("y er %d\n",y);
    return 0;
}
```

Vi kompilerer og kjører:

```
$ gcc -Wall printfdemo.c -o printfdemo
<vi får noen advarsler, men det kompilerer>
$ ./printfdemo
Tall er 3 og 1359008880.
y er 24
```

HVA? Ja, vi skrev verdien 24 til variabelen `y` (tallet 1359008880 ble skrevet ut for `y` i den første printf-setningen på grunn av `&`-operatoren, som gir adressen til variabelen). Så litt forenklet kan vi si at `%n` lar oss skrive til minnet, og `m$` lar oss kontrollere hvor i minnet vi skriver.

[4] I dette tilfellet setter vi en ampersand (`&`) foran variabelen `z`, fordi vi må gi minneadressen til variabelen i stedet for bare navnet. C har mange detaljer som dette, men vi bekymrer oss ikke om dem nå.

Til slutt: Den ekte format string-sårbarheten! Vi har nå sett at printf har noen interessante egenskaper (features), og ja, vi kaller dem features. Men features blir til *programvaresårbarheter* når brukere får en måte å utnytte dem på. Som forklart i det opprinnelige innlegget av Tim Newsham om Format String Attacks på Bugtraq-e-postlista i september 2000, er den mest opplagte feilen å gjøre hvis programmereren skal skrive ut en streng `str` som brukeren kontrollerer, og bestemmer seg for å gjøre
`printf(str);` UTRYGT!
i stedet for
`printf("%s", str);` TRYGT!
Hvis programmereren lar brukeren kontrollere argumentene til printf, kan de misbruke dette. F.eks. i følgende kode skal det ikke være mulig å få programmet til å skrive ut "Welcome!":

```
#include <stdio.h>
int login_OK = 0;
int main(void) {
    char password[256];

    printf("Enter password: ");
    fgets(password, sizeof(password), stdin);
    printf(password); /* VULNERABILITY */

    if ( login_OK == 1 ) {
        printf("Welcome!");
    }
    return 0;
}
```

Her kan en "hackerprøve ulik input og muligens konstruere en input med egenskapene vi har beskrevet over, for å overskrive `login_OK`-variabelen og dermed misbruke programmet.

Merk at dette ikke er noen fantastisk oppdagelse, vi trenger bare å lese man-sidene. `man 3 printf` forteller oss:

> Code such as printf(foo); often indicates a bug, since foo may contain a % character. If foo comes from untrusted user input, it may contain %n, causing the printf() call to write to memory and creating a security hole.

Hvis du ønsker å vite alle de ganske avanserte detaljene i format string-angrep, anbefaler jeg å se de utmerkede videoene fra Fabian Faessler (aka LiveOverflow) (i tillegg til å lese Bugtraq-innlegget nevnt over og printf-man-siden, selvfølgelig).

5.3 Kommandolinjeargumenter

For å gi kommandolinjeargumenter til et C-program, endrer vi
`int main(void)` til
`int main(int argc, char *argv[]))`

- `argc` inneholder antall argumenter som er gitt, men det inkluderer programnavnet, så det har en verdi på minst én. Hvis du gir ett kommandolinjeargument, vil `argc` ha en verdi på to. Hvis du gir to kommandolinjeargumenter, vil `argc` ha en verdi på tre, og så videre.
- `argv` er en array som inneholder de faktiske argumentene som er gitt. Siden C ikke har sin egen streng-datatype, er `argv` en array av "adressertil char-arrayer som inneholder argumentene, men ikke bekymre deg hvis du ikke forstår hva det betyr – det vi trenger å vite er hvordan vi kan aksessere dem:
  - Skriv ut det første argumentet:
    `printf("%s\n",argv[1]);`
  - Kopier det første argumentet til en annen streng:
    `char *first = strdup(argv[1]);`

Et lite demoprogram som skriver ut to kommandolinjeargumenter:

```
#include<stdio.h>
#include<string.h>
int main(int argc, char *argv[]) {
    char *first  = strdup(argv[1]);
    char *second = strdup(argv[2]);
    printf("%d args, %s og %s.\n",argc-1,first,second);
    return 0;
}
```

Vi kompilerer og kjører:

```
$ gcc -Wall argtest.c -o argtest
$ ./argtest solan reodor
2 args, solan og reodor.
```

Legg merke til at vi ikke bekymrer oss om robusthet i vår lille introduksjon til C, så hvis vi ikke gir nøyaktig to kommandolinjeargumenter til dette programmet, vil det aksessere variabler som ikke eksisterer, og dette fører typisk til et minneaksessfeil kalt segmentation fault":

```
$ ./argtest solan
Segmentation fault (core dumped)
```

5.4 Oppgaver

1. Hvordan er main-funksjonen i C forskjellig fra andre funksjoner?
2. Skriv kommandolinjen for å kompilere et C-program kalt `solan.c` til en kjørbar fil `solan`. Inkluder opsjonen for å aktivere alle advarsler i kompileringsprosessen.
3. Anta `int x=2, y=3;`, hva blir outputen av de følgende printf-setningene?
   (a) `printf("%d",x);`
   (b) `printf("%d\n%d\n",x,y);`
   (c) `printf("%2$d og %1$d\n",x,y);`
   (d) `printf("%2$d og %2$d\n",x,y);`
4. Forklar så detaljert du kan problemet med dette C-programmet:

```
#include<stdio.h>
int main(int argc, char *argv[]) {
    printf(argv[1]);
    return 0;
}
```

5. Last ned fila `mysil` og gjør den kjørbar:
   `curl -O https://erikhje.folk.ntnu.no/mysil; chmod 755 mysil`
   Finn ut om den har en format string-sårbarhet.
6. Syntaksen for en if-setning i C er
   `if ( <condition> ) { <code> } else { <code> }`
   Skriv et C-program `solan.c` som returnerer 1 hvis programmet har færre enn to kommandolinjeargumenter, og returnerer 0 ellers.
7. Skriv en Bash-kommandolinje med `if`-test som sier "Yes!"hvis exit-koden til solan (den kjørbare versjonen av programmet du skrev over) er null.
8. Husker demoprogrammet `argtest` fra avsnittet om kommandolinjeargumenter? Det krasjet med segmentation fault"hvis det fikk færre enn to argumenter, fordi det leste `argv[1]` og `argv[2]` uten å sjekke at de fantes. Gjør programmet *robust*: legg til en `if`-test øverst i `main` som skriver en kort bruksanvisning og avslutter med `return 1;` hvis `argc` er mindre enn 3. Bekreft at programmet nå gir en pen feilmelding i stedet for å krasje. (Merk at vi skriver `printf("Bruk: %s\n", argv[0])` og ikke `printf(argv[0])` – du husker hvorfor fra format string-avsnittet!)