5. Kontrolstrukturer - Motivation og Oversigt

I dette og de følgende afsnit vil vi - vigtigst af alt - møde forgreninger og løkker. Sådanne kaldes kontrolstrukturer, fordi vi med disse kan kontrollere måden hvorpå 'programpegepinden' bevæger sig rundt i programmet. Vi vil også diskutere sekvens og sammensætning.

5.1. Oversigt over kommandoer

Lektion 2 - slide 2

Vi har i de forrige afsnit mødt forskellige kommandoer, som opsummeres herunder. Kontrolstrukturerne, som er fokus i dette og efterfølgende afnit, virker på kommandoer. Med andre ord, kontrolstrukturerne angiver hvorledes en række kommandoer udføres i forhold til hinanden.

Vi starter med en oversigt over de kommandoer vi har mødt indtil nu

- Assignments
  - Udtryk med en assignment operator efterfulgt af semicolon
- **`Printf`** og **`scanf`** kommandoerne
  - Disse er reelt funktionskald efterfulgt af semicolon
- Den tomme kommando
  - Blot et semicolon

5.2. Motivation: Primitiv kontrol med hop

Lektion 2 - slide 3

Vi vil motivere os selv for at studere moderne kontrolstrukturer ved at se på en primitiv kontrolstyring med goto kommandoen.

En goto kommando tillader hop fra ét sted i programmet til et andet.

Programmer med goto kommandoer kan være *meget vanskelige af forstå*.

Nedenstående program tildeler værdien 2 til variablen `pos` (og `res` bliver `j`) hvis `i` er mindre eller lig med `j`. Hvis `i` er større end `j` tildeles `pos` værdien 1, og `res` bliver `i`. De blå programdele i program 5.1 kaldes for labels.

Det er forholdsvis svært at følge kontrolforløbet af selv dette simple program, som bruger betinget og ubetinget flytning af kontrolpunktet med goto kommandoer. Vi vender tilbage til en meget bedre udgave af program 5.1 i program 8.2

```
#include <stdio.h>

int main(void) {

  int i, j, pos, res;
  printf("Enter two integers: ");
  scanf("%d %d", &i, &j);

  if (i <= j) goto p1;
  pos = 1; res = i;
  goto p2;
  p1: pos = 2; res = j;

  p2: printf("pos: %d, res: %d\n", pos, res);

  return 0;
}
```

Program 5.1 *Et helt C program med to goto kommandoer.*

I næsten alle programmeringssituationer anbefales det at bruge kontrolstrukturer i stedet for goto kommandoer

Goto kommandoer er kun acceptable i undtagelsessituationer, eksempelvis for at komme ud af en 'dyb løkke'.

5.3. Oversigt over konstrolstrukturer

Lektion 2 - slide 4

Inden for kaster os over detaljerede kontrolstrukturer giver vi en oversigt over forskellige slags kontrolstrukturer.

De grundlæggende kontrolstrukturer kan klassificeres som sekventielle, sammensættende, udvælgende og gentagende

En *kontrolstruktur* styrer og kontrollerer rækkefølgen af udførelsen af et antal kommandoer

- **Sekventiel kontrol**
  - Kommandoer følger efter hinanden i den angivne rækkefølge
  - `kommando1; kommando2;`
- **Sammensætning**
  - En antal kommandoer sammensættes til én enkelt kommando
  - `{kommando1; kommando2;}`
- **Udvælgelse**
  - Én kommando udvælges blandt flere mulige.
  - Udvælgelsen foretages typisk ud fra værdien af et logisk udtryk.
  - `if (logiskUdtryk) kommando1; else kommando2;`

- **Gentagelse**
    - En kommando gentages et antal gange.
    - Antallet af gentagelser styres typisk af et logisk udtryk
    - `while (logiskUdtryk) kommando;`

Som sagt er udvælgende og gentagende kontrolstrukturer meget vigtige for langt de fleste af de programmer vi skriver. Sekventiel kontrol medtages for at få et komplet billede. Sammensætning kaldes også undertiden for *aggregering*. Det at *sammensætte dele til helheder* vil vi se i mange forskellige sammenhænge.

6. Logiske udtryk

Logiske udtryk bruges til at styre udvælgelse og gentagelse. Derfor er det vigtigt at vi har god styr på logiske udtryk. Dette vil vi få i dette afsnit.

6.1. Logiske udtryk

Lektion 2 - slide 6

Logiske udtryk kan naturligt sammenlignes med aritmetiske udtryk. Aritmetiske udtryk beregnes til talværdier. Logiske udtryk beregnes til sandhedsværdier, enten *sand* eller *falsk* (*true* eller *false* på engelsk). På samme måde som vi har set aritmetiske operatorer vil i dette afsnit studere logiske operatorer, som returnerer sandhedsværdier.

Vi skal vænne os til at håndtere sandhedsværdier på samme måde som vi håndterer tal, tegn, mv. Vi kan putte sandhedsværdier i variable, vi kan sammenligne dem mv. For en del af jer tager det måske lidt tid inden I finder den rette og naturlige melodi for håndteringen af sandhedsværdier.

Et *logisk udtryk* beregnes til en værdi i datatypen boolean.

Datatypen *boolean* indeholder værdierne *true* og *false*.

Her følger en række eksempler på logiske udtryk, som alle anvender heltalsvariablen `n`. Eksemplerne er vist inden for rammerne af et komplet C program i program 6.2.

```
int n = 6;

n == 5
n >= 5
n != 5
n >= 5 && n < 10
!(n >= 5 && n < 10)
n < 5 || n >= 10
```

Program 6.1 *Et uddrag af et C program med logiske udtryk.*

Først tester vi om `n` er lig med 5. `==` er altså en lighedoperator, som her returnerer false, idet `n` har værdien 6. Bemærk den afgørende forskel på assignment operatoren `=` og lighedsoperatoren `==`. I anden linie tester vi om `n` er større end eller lig med 5. Dette er sandt. I tredje linie tester vi om `n` er forskellig fra 5. Dette er også sandt. I fjerde line tester vi om `n` er større eller lig med 5 og mindre end 10. Bemærk `&&` svarer til 'logisk and'. (Se afsnit 6.2). Værdien er true igen. Udråbstegnet foran det tilsvarende udtryk betyder 'not', og det negerer altså værdien true til false. Det sidste logiske udtryk fortæller hvorvidt `n` er mindre end 5 eller større eller lig med 10. Når `n` stadig er 6 er dette naturligvis falsk. `||` betyder altså 'logisk eller'.

```
#include <stdio.h>

int main(void) {

  int n = 6, b;

  printf("%d\n", n == 5);

  if (n >= 5) printf("stor\n"); else printf("lille\n");

  b = n != 5; printf("%d\n", b);

  while (n >= 5 && n < 10) n++;
  printf("%d\n", n);

  if ( !(n >= 5 && n < 10) == (n < 5 || n >= 10))
    printf("OK\n");
  else printf("Problems\n");

  return 0;
}
```

Program 6.2 *De boolske udtryk vist i realistiske sammenhænge i et C program.*

Nedenstående er en vigtig detalje i C. Nul står for *false*, og ikke-nulværdier står alle for *true*.

I C repræsenteres *false* af en nulværdi, og *true* af en vilkårlig ikke-nulværdi

6.2. Sandhedstabeller

Lektion 2 - slide 7

Nu hvor vi er igang med at diskutere logiske udtryk benytter vi lejligheden til præcise definitioner af negering (not), konjunktion (and) og disjunktion (or) via sandhedstabellerne i hhv. tabel 6.1, tabel 6.2 og tabel 6.3. Negering håndteres af `!` operatoren i C. Tilsvarende hedder and operatoren `&&` i C, og or operatoren hedder `||`.

Da der kun findes to forskellige sandhedsværdier er det let at tabellægge alle forskellige muligheder for input og output af de nævnte operatorer. Bemærk her forskellen til heltal. Da der findes uendelig mange forskellige heltalsværdier vil tilsvarende definitioner af `+`, `-`, `*` og `/` være umulig.

A	!A
true	*false*
false	*true*

Tabel 6.1 *Sandhedstabel for not operatoren*

A	B	A && B	A || B
true	*true*	*true*	*true*
true	*false*	*false*	*true*
false	*true*	*false*	*true*
false	*false*	*false*	*false*

Tabel 6.2 *Sandhedstabel for and operatoren*

A	B	!(A && B)	!A || !B
true	*true*	*false*	*false*
true	*false*	*true*	*true*
false	*true*	*true*	*true*
false	*false*	*true*	*true*

Tabel 6.3 *Sandhedstabel for or operatoren*

Bemærk her at tabel 6.3 faktisk beviser DeMorgans boolske lov, som på et mere matematisk sprog siger at

- **not** (A **and** B) = **not** (A) **or not** (B).

I slide materialet viser vi en særlig udgave af operatorprioriteringstabellen i C, hvor de logiske og de sammenlignende operatorer er fremhævet. Se slide siden.

6.3. Short Circuit evaluering

Lektion 2 - slide 9

Vi vil nu beskæftige os med nogle binære, logiske operatorer, som under nogle omstændigheder ikke vil beregne værdien af den højre operand.

Direkte oversat 'kortsluttet beregning'

Short circuit evaluering betegner en beregning hvor visse operander ikke evalueres, idet deres værdi er uden betydning for resultatet

- `x && y`
  - y beregnes kun hvis x er *true*
- `x || y`
  - y beregnes kun hvis x er *false*

6.4. Short Circuit evaluering: eksempler

Lektion 2 - slide 10

Vi ser først på et eksempel, som er ganske kunstig, men dog illustrativ. Dernæst, i program 6.6 følger et 'real life' eksempel.

Vi viser eksempler på C programmer med short circuit evaluering

Herunder ses en række assignment operatorer som indgår i logiske udtryk. Dette kan lade sig gøre i C idet assignment operatorer returnerer en værdi, ligesom alle andre operatorer. Dette diskuterede vi i kapitel 3. En anden essentiel observation for nedenstående er at tallet nul spiller rollen som sandhedsværdien *false*, og at andre tal spiller rollen som *true*. Dette blev diskuteret i afsnit 6.1.

```
#include <stdio.h>

int main(void) {

  int i = 0, j = 0;

  (i = 1) && (j = 2);
  printf("%d %d\n", i, j);

  (i = 0) && (j = 3);
  printf("%d %d\n", i, j);

  i = 0 || (j = 4);
  printf("%d %d\n", i, j);

  (i = 2) || (j = 5);
  printf("%d %d\n", i, j);

  return 0;
}
```

Program 6.3 *Short circuit beregning i kombination med assignments.*

De to første assignments udføres begge, idet værdien af den første er 1 (true). Af de to næste assignment udføres kun det første. Årsagen er at `i` assignes til 0, som er false, hvilket forårsager undertrykkelse af beregningen af højre operand af &&. Det næste assignment skal fortolkes som `i = (0 || (j = 4))`. Husk igen af 0 er false. Højre operand af `||` skal derfor beregnes, jf. reglerne fra afsnit 6.3. Endelig gennemføres assignmentet til `j` ikke i sidste røde linie, idet venstre operand beregnes til 2 (true).

Med ovenstående observationer er program 6.4 ækvivalent med program 6.3.

```
#include <stdio.h>

int main(void) {

  int i = 0, j = 0;

  (i = 1) && (j = 2);
  printf("%d %d\n", i, j);

  (i = 0);
  printf("%d %d\n", i, j);

  i = 0 || (j = 4);
  printf("%d %d\n", i, j);

  (i = 2);
  printf("%d %d\n", i, j);

  return 0;
}
```

Program 6.4 *Et ækvivalent program.*

Outputtet fra de to ovenstående (ækvivalente) programmer ses her:

```
1 2
0 2
1 4
2 4
```

Program 6.5 *Output fra ovenstående programmer.*

Og nu til et mere dagligdagsprogram. I den røde linie ses et logisk udtryk, hvor den venstre operator beskytter den højre. Det skal forstås således at `sqrt(x)` kun bliver beregnet hvis `x` er ikke-negativ. Dette er helt essentielt for at undgå en regnefejl i kvadratrodsfunktionen `sqrt`, som jo kun kan tage ikke-negative parametre.

```
#include <math.h>
#include <stdio.h>

double sqrt(double);

int main(void) {

  double x;

  printf("Enter a number: ");
  scanf("%lf", &x);

  if (x >= 0 && sqrt(x) <= 10)
    printf("x is positive, but less than or equal to 100.0\n");
  else printf("x is negative or greater than 100.0\n");

  return 0;
}
```

Program 6.6 *Short circuit beregning som beskyttelse mod fejlberegning.*

Prøvekør meget gerne programmet. Bemærk at på nogle C systemer skal der en ekstra option **`-lm`** på kompileren for at håndtere kvadratrodsfunktionen.

7. Sammensætning af kommandoer

Af hensyn til en ren strukturering efter sammensætning, udvælgelse og gentagelse har vi her et kort afsnit om sammensætning af flere kommandoer til én helhedskommando - blokke.

7.1. Sammensætning

Lektion 2 - slide 12

Vi bruger ofte betegnelsen *blokke* for en sammensætning af kommandoer til én kommando. I en blok af kommandoer kan vi endvidere frit erklære variable. Vi foretrækker dog at variable introduceres i starten af blokken, altså før den første kommando.

En sammensat kommando, også kaldet en *blok*, er en gruppering af kommandoer til én kommando, hvori der i starten kan forekomme erklæringer af variable

Bestanddelene i en sammensat kommando indhegnes af 'tuborg klammer':

```
{declaration-or-command-list}
```

Syntaks 7.1 *En blok*

Det er ofte nødvendig at bruge sammensatte kommandoer, nemlig på pladser hvor kun én enkelt kommando er tilladt. Dette er f.eks. tilfældet for kroppene af if-else og while, som vi ser på i hhv. afsnit 8.1 og afsnit 9.1.

> Blokke er ofte nødvendige i forbindelse med udvælgende og gentagende kontrolstrukturer i C.
>
> Blokke kan indlejres i hinanden.

Vi viser herunder et eksempel på indlejring af blokke i hinanden. Den yderste blok er en fast bestanddel af `main` funktionen. I afsnit 20.1 nærmere betegnet program 20.1, vender vi tilbage til det faktum, at de samme navne `a`, `b` og `c` erklæres i de indlejrede blokke.

```
#include <stdio.h>

int main(void) {

  int a = 5, b = 7, c;

  c = a + b;

  {
    int a = 15, b = 17, c;

    {
      int a = 25, b = 27, c;
      c = a + b;
      printf("Inner: c er %d\n", c);
    }

    c = a + b;
    printf("Middle: c er %d\n", c);
  }

  c = a + b;
  printf("Outer: c er %d\n", c);

  return 0;
}
```

Program 7.1 *Tre indlejrede blokke med lokale erklæringer af heltal.*

8. Udvælgelse af kommandoer

Der er kun få programmer som slavisk følger ét enkelt spor uden forgreninger. Med andre ord er der kun få programmer der udelukkende betjener sig af sekventiel og sammensat kontrol, som beskrevet i hhv. afsnit 5.3 og afsnit 7.1.

Langt de fleste programmer skal kunne reagere forskelligt på en række hændelser. Derfor er *udvælgelse* - også kaldet *selektion* - af kommandoer et vigtigt emne. Vi studerer udvælgelse i dette kapitel.

8.1. Udvælgelse med if (1)

Lektion 2 - slide 14

Den mest basale kontrolstruktur til udvælgelse kaldes **if-then-else**. I den rene form, som i C afspejles af syntaks 8.1, vælges der mellem netop to muligheder. Valget styres af et logisk udtryk, som jo netop har to mulige udfald: *true* eller *false*. Som det fremgår af syntaksboken anvender C ikke nøgleordet **then**, og det er derfor lidt misvisende af tale om **if-then-else** kontrolstrukturer i C. Vi vil derfor blot tale om **if-else** strukturer.

> En *selektiv kontrolstruktur* udvælger én kommando til udførelse blandt en mængde af muligheder

```
if (logicalExpression)
  command1
else
  command2
```

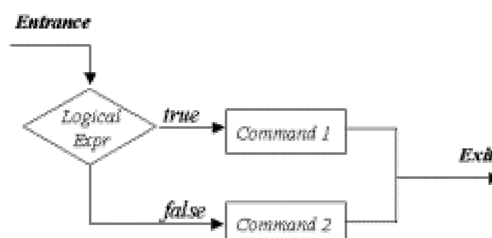
Syntaks 8.1 *En if-else kontrolstruktur.*

Bemærk parenteserne omkring *logicalExpression*, som altid skal være der. Parenteserne er en del af syntaksen for if-else i C. Tilsvarende forhold gør sig gældende for switch (syntaks 8.3), while (syntaks 9.1), og do (syntaks 9.2). Som et specialtilfælde kan else-delen være tom. I så fald er det ikke nødvendig at skrive nøgleordet 'else'. Vi taler i dette tilfældet om en *betinget kommando*. Vi siger altså at *command* er betinget af det logiske udtryk *logicalExpression* i syntaks 8.2. *Command* udføres kun hvis værdien af det logiske udtryk er sandt.

```
if (logicalExpression)
  command
```

Syntaks 8.2 *En betinget kommando.*

Herunder viser vi en kontrol flowgraf for en if-then-else kommando, som modsvarer syntaks 8.1. Sådanne grafer er nyttige til illustration er enkelte kontrolstrukturer i isolation, men de bliver dog for det meste uoverskuelige hvis vi forsøger at anvende dem på et helt program.



Figur 8.1 *Flow graf for if*

8.2. Udvælgelse med if (2)
Lektion 2 - slide 15

Vi fortsætter vores diskussion af if-else i dette afsnit. Betydningen af syntaks 8.1 kan sammenfattes i følgende punkter:

- Betydning af if-else:
  - Først beregnes det logiske udtryk
  - Hvis værdien er sand udføres command1
  - Hvis værdien er falsk udføres command2

Det er nu tid til at se på eksempler. I program 8.1 starter vi med et program, der finder og udskriver det største af to indlæste tal x og y. Bemærk hvordan vi i den røde del styrer kontrollen til enten at udføre assignmentet `max = y` eller `max = x`, afhængig af udfaldet af sammenligningen mellem `x` og `y`.

```
#include <stdio.h>

int main(void) {

  double x, y, max;

  printf("Enter two real numbers: ");
  scanf("%lf %lf", &x, &y);

  /* find the maximum of x and y */
  if (x < y)
    max = y;
  else
    max = x;

  printf("The maximum of x and y is %12.8f\n", max);

  return 0;
}
```

Program 8.1 *Et program som beregner det største af to doubles.*

Hernæst følger et andet eksempel, som vi allerede har varmet op i afsnit 5.2, nærmere betegnet i program 5.1. Her er tale om en omskrivning af goto programmet til at bruge en if-else kontrolstruktur.

Sammenlign nøje med program 5.1 og overbevis dig om if-else programmets bedre struktur og større læsbarhed.

```c
#include <stdio.h>

int main(void) {

  int i, j, pos, res;
  printf("Enter two integers: ");
  scanf("%d %d", &i, &j);

  if (i <= j){
    pos = 2; res = j;}
  else {
    pos = 1; res = i;}

  printf("pos: %d, res: %d\n", pos, res);

  return 0;
}
```

Program 8.2 *Goto programmet reformuleret med en if-else kontrolstruktur.*

Brug af kontrolstrukturer ala **`if-else`** er udtryk for *struktureret programmering*

Struktureret programmering var en af de store landvindinger inden for faget i 1970'erne.

8.3. 'Dangling else' problemet

Lektion 2 - slide 16

I dette afsnit vil vi beskrive en speciel sammensætning af if og if-else kontrolstrukturer, som foranlediger et tvetydighedsproblem.

Hvis **`if`** og **`if-else`** kontrolstrukturer indlejres i hinanden kan der opstå tvetydighedsproblemer

Problemet illustreres i eksempel program 8.3 herunder.

I den røde del signalerer indrykningen at else delen hører til den inderste if. I situationen hvor `a` er 3 og `b` er 4 bliver der ikke printet noget som helst i den røde del.

På trods af indrykningen i den blå del, er den blå del helt ækvivalent med den røde del. Som opsummeret herunder gælder der, at en else del altid knytter sig til den nærmeste if. Der skrives altså heller ikke noget ud i den blå del.

I den lilla del har vi indlejret `if (b==2) print(...)` i en blok, som er angivet med brune brackets. Dermed knyttes else delen til den yderste if. I denne situation udskrives et stort F.

```
#include <stdio.h>

int main(void) {

  int a = 3, b = 4;

  if (a == 1)
    if (b == 2)
      printf("A\n");
    else
      printf("B\n");

  if (a == 1)
    if (b == 2)
      printf("C\n");
  else
    printf("D\n");

  if (a == 1){
    if (b == 2)
      printf("E\n");}
  else
    printf("F\n");

  return 0;
}
```

Program 8.3 *Illustration af dangling else problemet.*

Som beskrevet ovenfor vil program 8.3 blot udskrive et stort F.

> En `else` knytter sig altid til den inderste `if`.
>
> Det kan anbefales at bruge sammensætning med {...} hvis man er i tvivl om fortolkningen.

8.4. Udvælgelse med switch

Lektion 2 - slide 17

Switch er en anden kontrolstruktur til udvælgelse. Med switch vælger man mellem et vilkårligt antal muligheder. Valget foregår på basis af en heltallig værdi. Derfor er en switch ikke så kraftfuld som evt. indlejrede if-else konstruktioner. Mere om dette i afsnit 8.5.

```
switch (expression) {
  case const1: command-list1
  case const2: command-list2
  ...
  default: command-list
}
```

Syntaks 8.3 *Opbygningen af en switch kontrolstruktur.*

Vi beskriver betydningen af switch i de følgende punkter

- Udtrykket beregnes - skal være et heltal eller heltalsagtigt
- Hvis værdien svarer til en af konstanterne, flyttes kontrollen til den tilsvarende kommando
- Hvis værdien ikke svarer til en af konstanterne, og hvis der er en default, flyttes kontrollen til default kommandoen
- Hvis værdien ikke svarer til en af konstanterne, og hvis der ikke er en default, flyttes kontrollen til afslutningen af switchen

Der er ét væsentlig forhold, som skal bemærkes. Når et af 'casene' i en switch er udført, hopper man ikke automatisk til enden af switch konstruktionen. Derimod fortsætter udførelsen af de efterfølgende cases i switchen. Som regel er dette ikke hvad vi ønsker. Som illustreret i program 8.4 kan man afbryde dette 'gennemfald' med en break i hvert case. Dette er de blå dele i program 8.4.

```
#include <stdio.h>

int main(void) {

  int month, numberOfDays, leapYear;

  printf("Enter a month - a number between 1 and 12: ");
  scanf("%d", &month);
  printf("Is it a leap year (1) or not (0): ");
  scanf("%d", &leapYear);

  switch(month){
    case 1: case 3: case 5: case 7: case 8: case 10: case 12:
      numberOfDays = 31; break;
    case 4: case 6: case 9: case 11:
      numberOfDays = 30; break;
    case 2:
      if (leapYear) numberOfDays = 29;
      else numberOfDays = 28; break;
    default: exit(-1);  break;
  }

  printf("There are %d days in month %d\n", numberOfDays, month);

  return 0;
}
```

Program 8.4 *Et program der beregner antal dage i en måned.*

Programmet ovenfor finder antallet af dage i en given måned. Hvis måneden er 1, 3, 5, 7, 8, 10 eller 12 er der altid 31 dage. Hvis måneden er 4, 6, 9 eller 11 er der 30 dage. I februar, den 2. måned, er der 28 eller 29 dage afhængig af om året er et skudår eller ej.

Vi har mere at sige om break i afsnit 9.7.

Vi ønsker næsten altid at afslutte et tilfælde i en switch med en **`break`**

8.5. If-else kæder

Lektion 2 - slide 18

Vi kan kun bruge switch til multivejsudvælgelse i det tilfælde at valget kan foretages på baggrund af en simpel, heltallig værdi.

I mange tilfælde er det nødvendigt at foretage en multivejsudvælgelse ud fra evaluering af en række betingelser. Når der er behov for dette kan vi bruge if-else kæder. Bemærk, at if-else kæder laves ved at indlejre en if-else kontrolstruktur i else-delen af en anden if-else struktur.

En **`if-else`** kæde er en **`if-else`** kontrolstruktur, hvor else-delen igen kan være en **`if-else`** struktur

Som et eksempel viser vi et program som omregner procentpoint til en karakter på 13-skalaen. Omregningen svarer til den officielle omregningstabel for skriftligt arbejde inden for naturvidenskab.

Det vil være meget akavet at skulle udtrykke omregningen med en switch. Dette vil nemlig kræve en case for hvert heltal, hvilket naturligvis er helt uacceptabelt. Det er langt mere naturligt at formulere logiske udtryk for de forskellige udfald.

```
#include <stdio.h>

int main(void) {

  int percent, grade;

  printf("How many percent? ");
  scanf("%d",&percent);

  if (percent >= 90)
    grade = 11;
  else if (percent >= 82)
    grade = 10;
  else if (percent >= 74)
    grade = 9;
  else if (percent >= 66)
    grade = 8;
  else if (percent >= 58)
    grade = 7;
  else if (percent >= 50)
    grade = 6;
  else if (percent >= 40)
    grade = 5;
  else if (percent >= 10)
    grade = 3;
```

```
  else grade = 0;

  printf("%d percent corresponds to the Danish grade %d\n\n",
          percent, grade);

  return 0;
}
```

Program 8.5 *Et program med en if-else kæde der beregner beregner en karakter ud fra et antal procentpoint.*

Vi har i program 8.5 anvendt en speciel indrykning af if-else strukturerne. En mere almindelig indrykning, som understreger måden hvorpå if-else er indlejret i else dele, er vist i program 8.6 herunder. Med denne indrykning havner vi dog hurtigt uden for den højre margin, og af denne årsag foretrækkes den flade indrykning i program 8.5.

```
#include <stdio.h>

int main(void) {

  int percent, grade;

  printf("How many percent? ");
  scanf("%d",&percent);

  if (percent >= 90)
   grade = 11;
  else if (percent >= 82)
        grade = 10;
       else if (percent >= 74)
             grade = 9;
            else if (percent >= 66)
                  grade = 8;
                 else if (percent >= 58)
                       grade = 7;
                      else if (percent >= 50)
                            grade = 6;
                           else if (percent >= 40)
                                 grade = 5;
                                else if (percent >= 10)
                                      grade = 3;
                                     else grade = 0;

  printf("%d percent corresponds to the Danish grade %d\n\n",
          percent, grade);

  return 0;
}
```

Program 8.6 *Alternativt og uønsket layout af if-else kæde.*

Læs mere om indrykning og 'style' i afsnit 3.20 side 106-108 i *C by Dissection*.

En **if-else** kæde kan generelt ikke programmeres med en `switch` kontrolstruktur

8.6. Den betingede operator

Lektion 2 - slide 19

If-else og switch er kontrolstrukturer, som styrer udførelsen af kommandoer. I mange engelske bøger bruges betegnelsen *statements* for kommandoer. I dette afsnit vender vi tilbage til udtryk, som vi studerede allerede i afsnit 2.1. Helt konkret ser vi på en if-then-else konstruktion som har status af et udtryk i C.

> I C findes en betinget operator med med tre operander som modsvarer en **`if-else`** kontrol struktur.
>
> En forekomst af en betinget operator er et udtryk, hvorimod en forekomst af **`if-else`** er en kommando.

Syntaksen af operatoren, som kaldes `?:`, er vist herunder. Ligesom if-else har den tre bestanddele, nemlig et logisk udtryk og to andre betanddele, som her begge er vilkårlige udtryk. Begge disse udtryk skal dog være af samme type.

```
logicalExpression ? expression1 : expression2
```

Syntaks 8.4 *Opbygningen af den betingede operator i C.*

Bemærk at den betingede operator tager tre operander i modsætning til de fleste andre operatorer, som enten er unære eller binære.

Her giver vi betydningen af `?:` operatoren relativ til syntaks 8.4.

- Først beregnes *logicalExpresssion*
- Hvis værdien er sand beregnes *expression1* ellers *expression2,* hvorefter den pågældende værdi returneres af **`?:`** operatoren

Vi viser herunder et eksempel på brug af den betingede operator. Givet et indlæst heltal i variablen `x` bestemmer det røde udtryk fortegnet af `x`. Det er enten strengen "negativ", "neutral", eller "positiv".

Ifølge tabel 2.3 associerer den betingede operator fra højre mod venstre. (Du finder den på prioritetsniveau 3 i tabellen). Det betyder at strukturen i udtrykket er følgende:

- `"x < 0 ? "negative" : (x == 0 ? "neutral" : "positive")`

```
#include <stdio.h>

int main(void) {

  int x;

  printf("Enter an integer: ");
  scanf("%d", &x);

  printf("The sign of %d is %s\n\n",
         x,
         x < 0 ? "negative" : x == 0 ? "neutral" : "positive");

  return 0;
}
```

Program 8.7 *Et program der bestemmer fortegnet af et tal.*

Bemærk at det ikke ville have været muligt at have en if-else konstruktion inden i kaldet af printf. Kun udtryk kan optræde som parametre til funktioner. Og den røde del af program 8.7 er netop et udtryk.

Hermed slutter vores behandling af kontrolstrukturer og udtryk til udvælgelse. I næste afsnit ser vi på gentagelse.

9. Gentagelse af kommandoer

En af styrkerne ved en computer er dens evne til at gennemføre mange gentagelser på kort tid. Når vi programmerer bruger vi forskellige løkker til at kontrollere gentagelser af kommandoer. Undertiden bruget vi også betegnelsen *iterationer* i stedet for gentagelser.

I C er der tre forskellige løkker, som vi alle vil se på i dette afsnit. Vi starter med whileløkker.

9.1. Gentagelse med while (1)

Lektion 2 - slide 21

Hvis man kun vil lære én løkke konstruktion er det while løkken man bør kaste sig over. Som vi vil indse i afsnit 9.8 kan alle løkker i C relativt let omformes til whileløkker.

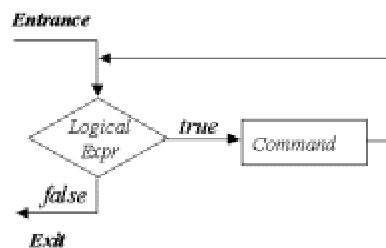
I en løkke er det muligt at gentage en kommando nul, én eller flere gange.
I en while løkke er det ubestemt hvormange gange kommandoen udføres.

Syntaksen af whileløkker minder om syntaksen af if, jf. syntaks 8.2 . Betydningen af en while kontrolstruktur er derimod helt forskellig fra en if kontrolstruktur.

```
while (logicalExpression)
  command
```

Syntaks 9.1 *Syntaksen af en while løkke i C.*

Vi viser en flowgraf for whileløkken herunder. Bemærk for det første muligheden for at køre rundt i løkken, altså det at kommandoen kan udføres et antal gange, under kontrol af det logiske udtryk. Bemærk også, at vi kan udføre whileløkken helt uden af udføre kommandoen (nemlig i det tilfælde at det logiske udtryk er falsk når vi møder det første gang).



Figur 9.1 *Flow graf for en while løkke*

9.2. Gentagelse med while (2)

Lektion 2 - slide 22

Vi beskriver her på punktform betydningen af en whileløkke.

- Betydningen af while:
  - Det logiske udtryk beregnes
  - Hvis værdien er *true* udføres kommandoen, og while løkken startes forfra
  - Hvis værdien er *false* afsluttes while løkken

Vores første eksempel er et klassisk program, nemlig *Euclids algoritme* til at finde den største fælles divisor af to positive heltal. Givet to heltal `small` og `large` ønsker vi altså at finde det største tal, som både går op i `small` og `large`.

```
#include <stdio.h>

int main(void) {
  int i, j, small, large, remainder;

  printf("Enter two positive integers: ");
  scanf("%d %d", &i, &j);

  small = i <= j ? i : j;
  large = i <= j ? j : i;

  while (small > 0){
    remainder = large % small;
    large = small;
    small = remainder;
  }

  printf("GCD of %d and %d is %d\n\n", i, j, large);

  return 0;
}
```

Program 9.1 *Euclids algoritme - største fælles divisor - programmeret med en while løkke.*

Ved første øjesyn er det ikke umiddelbart klart, hvorfor algoritmen virker. Man mener, at netop ovenstående algoritme er det første eksempel i historien på en ikke-triviel beregningsmæssig opskrift (algoritme).

Lad os kort forklare programmet ud fra et eksempel. Lad os antage at vi skal finde den største fælles divisor af tallene 30 og 106. Dette tal betegnes som gcd(30,106), og det viser sig at være 2. gcd betyder "greatest common divisor". Variablen `small` starter altså med at være 30 og `large` er 106. I det første gennemløb af whileløkken beregnes resten af division af 106 med 30. Dette er 16, og dette tal vil generelt altid være mindre end divisoren 30. Hermed bliver `large` sat til 30 og `small` til 16.

Set over et antal gennemløb producerer programmet talrækken

- 106, 30, 16, 14, 2, 0

Den essentielle indsigt er nu at gcd(106,30) = gcd(30,16) = gcd(16,14) = gcd(14,2) = gcd(2,0) = 2. Dette indses ved et induktionsargument. Den sidste lighed følger af, at det største tal der både går op i 2 og 0 naturligvis er 2.

Med dette vil programmet finde, at det største tal der både går op i 106 og 30 er 2. Ydermere viser det sig, at programmet finder resultatet i et bemærkelsesværdigt lille antal skridt.

Herunder er vist en simpel variant af program 9.1 som udskriver information om mellemresultaterne.

```
#include <stdio.h>

int main(void) {
  int i, j, small, large, remainder;

  printf("Enter two positive integers: ");
  scanf("%d %d", &i, &j);

  small = i <= j ? i : j;
  large = i <= j ? j : i;

  printf("%d %d ", large, small);

  while (small > 0){
    remainder = large % small;
    large = small;
    small = remainder;
    printf("%d ", small);
  }

  printf("\n\nGCD of %d and %d is %d\n\n", i, j, large);

  return 0;
}
```

Program 9.2 *En udgave af Euclids algoritme som udskriver den beregnede talrække.*

9.3. Gentagelse med do (1)

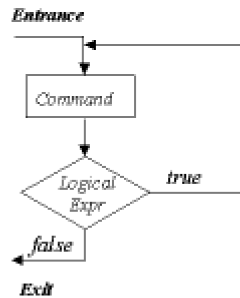
Lektion 2 - slide 23

I nogle situationer er vi interesserede i at arbejde med løkker, som sikrer mindst én gentagelse. I C er **do**-løkken af en sådan beskaffenhed. I andre sprog, ala Pascal, hedder sådanne løkker ofte **repeat until**.

En `do`-løkke sikrer altid mindst ét gennemløb af løkken

```
do
  command
while (logicalExpression)
```

Syntaks 9.2 *Syntaksen af en do-løkke i C*



Figur 9.2 *Flow graf for en do løkke*

9.4. Gentagelse med do (2)

Lektion 2 - slide 24

Ligesom for de andre kontrolstrukturer vi har set vil vi beskrive betydningen på punktform, og vi vil se på et eksempel.

- Betydningen af `do`:
    - Kommandoen udføres
    - Det logiske udtryk beregnes
    - Hvis værdien er *true* overføres kontrollen til starten af `do`
    - Hvis værdien er *false* afsluttes do løkken

I nedenstående program viser vi en programstump, som vi kunne kalde "ask user". Programmet insisterer på et ja/nej svar, i termer af bogstavet 'y' eller 'n'. Vi antager her, for at simplificere opgaven, at brugeren kun indtaster ét tegn.

```
#include <stdio.h>

int main(void) {

  char answer, forget;

  do {
    printf("Answer yes or no (y/n): ");
    scanf("%c%c", &answer, &forget);
  } while (answer != 'y' && answer != 'n');

  printf("The answer is %s\n\n",
          answer == 'y'? "yes" : "no");

  return 0;
}
```

Program 9.3 *Et program som ønsker et 'yes/no' svar.*

Kroppen af løkken skal udføres mindst én gang. Hvis ikke svaret er tilfredsstillende skal vi udføre kroppen endnu en gang.

Bemærk at vi læser to tegn, `answer` og `forget`. `forget` introduceres for læse lineskiftet (RETURN), som er nødvendig for at få scanf til at reagere. Dette er ikke en perfekt løsning på problemet, men dog bedre end hvis kun ét tegn læses. Leg gerne selv med mulighederne.

9.5. Gentagelse med for (1)

Lektion 2 - slide 25

Ideen med den næste løkke, for-løkken, er i bund og grund at kunne kontrollere et fast og *på forhånd kendt antal gentagelser*. Det er imidlertid sådan, at for-løkker i C er mere generelle end som så.

> I mange sprog benyttes **`for`** kontrolstrukturen hvis man på forhånd kender antallet af gentagelser.
>
> **`for`** kontrolstrukturen i C kan benyttes således, men den bruges også ofte til mere generel gentagelse.

Her følger syntaksen og dernæst den punktvise betydning af for-løkken.

```
for (initExpression; continueExpression; updateExpression)
  command
```

Syntaks 9.3 *Opbygningen af en for-løkke i C.*

- Betydning af en for-løkke:
  - Trin 1: *initExpression* evalueres af hensyn til sideeffekter
  - Trin 2: *continueExpression* evalueres, og hvis den er *false* (0) er **`for`** løkken færdig
  - Trin 3: **`for`** løkkens *command* udføres
  - Trin 4: *updateExpression* evalueres, og der fortsættes med trin 2

I nedenstående eksempel udskriver vi alle tal fra og med `lower` til og med `upper`. Dette eksempel egner sig godt til en for-løkke, idet vi inden løkken starter ved at der skal bruges `upper` - `lower` + 1 gentagelser.

```
#include <stdio.h>

int main(void) {

  int upper, lower, k;
```

```
  printf("Enter two integers, lower and upper: ");
  scanf("%d %d", &lower, &upper);

  for (k = lower; k <= upper; k++)
    printf("k = %d\n", k);

  return 0;
}
```

Program 9.4 *En typisk for løkke med et forudbestemt antal gentagelser.*

9.6. Gentagelse med for (2)

Lektion 2 - slide 26

Lad os illustrere den udvidede styrke af for-løkker i C. Altså styrken ud over at kunne kontrollere et på forhånd bestemt antal gentagelser.

> Det er muligt at styre en kompleks gentagelse som anvender flere kontrolvariable ved brug af en `for` løkke

Vi genprogrammerer Euclids Algoritme fra program 9.1, nu med brug af en for-løkke. Vi vil referere til de tre bestanddele af en for-løkke, jf. syntaks 9.3.

I den røde *initExpression* sørger vi for at variablene `sml` og `lg` initielt indeholder det største hhv. det mindste af de tal, som er indlæst. I den lilla *updateExpression* laver næste skridt i rækkeudviklingen, som vi beskrev i afsnit 9.2. I den blå *continueExpression* kontrollerer vi termineringen (afslutningen) af gentagelsen. Bemærk at alle tilstandsændringer og check foregår i init, continue og update delene. For løkken i program 9.5 er uden krop!

```
#include <stdio.h>

int main(void) {
  int i, j, sml, lg, rem;

  printf("Enter two positive integers: ");
  scanf("%d %d", &i, &j);

  for(sml = i<=j?i:j , lg = i<=j?j:i;
      sml > 0;
      rem = lg % sml , lg = sml, sml = rem);

  printf("GCD of %d and %d is %d\n\n", i, j, lg);

  return 0;
}
```

Program 9.5 *Euclids algoritme - største fælles divisor - programmeret med en for-løkke 'uden krop'.*

Bemærk brugen af komma i de røde og blå dele. Komma er her en operator, som beregner udtryk i sekvens, og som returnerer værdien af det sidst evaluerede udtryk. Kommaoperatoren minder om semikolon, som skal placeres mellem (eller rettere efter) kommandoer i C. Læs om kommaoperatoren i afsnit 3.13 side 98 af *C by Dissection*. Kommaoperatoren har lavest mulig prioritering, jf. tabel 2.3 hvilket betyder at udtryk med alle andre operatorer end komma beregnes først.

Alle udtryk kan udelades, hvilket afstedkommer en uendelig løkke

Vi skitserer herunder situationen hvor *initExpression* og *continueExpression* begge er tomme. Dette afstedkommer en uendelig løkke - altså en løkke som ikke terminerer med mindre vi udfører et eksplicit hop ud af for-løkkens krop.

```
#include <stdio.h>

int main(void) {

  long i = 0;

  for( ; ; ++i)
    printf("%ld: Hi there...\n", i);

  return 0;
}
```

Program 9.6 *En for løkke, der optæller en variabel uendeligt.*

9.7. Break, continue og return

Lektion 2 - slide 27

Vi vil her se på mulighederne for at påvirke kontrolforløbet af løkker og visse udvælgende kontrolstrukturer.

I C kan man bruge goto og labels, som antydet i afsnit 5.2. Vi foretrækker dog mere specifikke og specialiserede hop kommandoer, som ikke kræver at vi introducerer labels. Vi ser nu på **break**, **continue** og **return**.

Anvendelse af generelle hop - goto - anses for meget dårlig programmeringsstil.

Brug af 'mere specialiserede hop' bruges oftere.

- **break**
    - anvendes til at springe ud af (den inderste) switch, while, do, eller for
- **continue**
    - anvendes til at afslutte kommandoen i en while, do eller for løkke
- **return**
    - anvendes til at afbryde, og returnere et resultat fra en funktion

Vi viser her et program, som illustrerer continue og break. Programmet indeholder en 'uendelig for-løkke'. Den røde del sørger for at gentagelser med lige *i* værdier overspringes, i det mindste således at *i* ikke skrives ud i printf kommandoen. Den blå del sørger for at afbryde løkken når værdien af *i* bliver større end 1000.

Samlet set vil program 9.7 udskrive alle ulige tal mellem 1 og 1000.

```
#include <stdio.h>

int main(void) {

  long i = 1;

  for( ; ; ++i){
    if (i % 2 == 0) continue;
    if (i > 1000) break;
    printf("%ld: Hi there...\n", i);
  }

  return 0;
}
```

Program 9.7 *Illustration af break og continue i en for løkke.*

Vi møder return kommandoen i kapitel 10, hvor emnet er funktioner.

9.8. Konvertering af do og for løkker til while

Lektion 2 - slide 28

I dette afsnit overbeviser vi os om, at både do- og for-løkker kan omskrives til whileløkker. Skabelonen for omskrivningerne ses i tabel 9.1.

Ethvert program med **`do`** og **`for`** løkker kan let omskrives til kun at bruge **`while`** løkker

<table>
<tr><th>Original</th><th>Transformation</th></tr>
<tr><td><pre>for (expr1; expr2; expr3)
  command</pre></td><td><pre>expr1;
while (expr2) {
  command;
  expr3;
}</pre></td></tr>
<tr><td><pre>do
  command
while (expr)</pre></td><td><pre>command;
while (expr)
  command;</pre></td></tr>
</table>

Tabel 9.1

Læg mærke til, at der for alle gentagende kontrolstrukturer gælder, at *kontrol udtrykket* er en *fortsættelsesbetingelse*.

Med andre ord afbrydes alle former slags løkker i C når kontroludtrykkets værdi bliver *false*.