10. Funktioner - Lasagne

Nøgleordet i dette og de følgende afsnit er *abstraktion*. I C udgøres den primære abstraktionsmekanisme af funktioner.

Ideen i dette kapitel er at introducere og motivere abstraktioner og funktioner via analogier fra gastronomien. Programmer kan opfattes som *opskrifter på beregninger*. Vi taler i denne sammenhæng om algoritmer. I den gastronomiske verden kender vi også til opskrifter, nemlig *opskrifter på det at lave mad*.

Vi studerer således indledningsvis en lasagne opskrift. God appetit!

10.1. Lasagne al forno

Lektion 3 - slide 2

Herunder viser vi en opskrift på lasagne al forno. Vi kan karakterisere den som en sekvens af trin, som vi antager vil kunne udføres af en *mad fortolker*. Uheldigvis har vi ikke (endnu) en automatiseret fortolker, som kan eksekvere opskriften. Vi må give den til en kyndig person (en kok, f.eks.) for hvem de enkelte trin giver mening. Vi vil her antage, at de enkelte trin er tydelige og klare for den person eller maskine, som effekturerer opskriften til en varm og lækker ret.

- Bland og ælt 500 g durum hvedemel, 5 æg, lidt salt og 2 spsk olie.
- Kør pastaen gemmen en pastamaskine i tynde baner.
- Smelt 3 spsk smør, og rør det sammen med 3 spsk mel.
- Spæd med med 5 dl mælk og kog sovsen under svag varme.
- Svits 2 hakkede løg med 500 g hakket oksekød, samt salt og pebber.
- Tilsæt 3 spsk tomatpuré, en ds. flåede tomater og 3 fed knust hvidløg.
- Kog kødsovsen 10 minutter.
- Bland nu lagvis pastabaner, kødsovs og hvid sovs. Drys med paramesanost.
- Gratiner retten i ovnen 15 minutter ved 225 grader.

Alle som har lavet lasagne vil vide, at denne ret laves af kødsovs, hvid sovs, og pasta. Vi ser det således som en svaghed, at disse *abstraktioner* ikke har fundet indpas i opskriften. Indførelse af abstraktioner vil være vores næste træk.

10.2. Struktureret lasagne

Lektion 3 - slide 3

Struktureret lasagne er nok ikke på menukortet i mange restauranter. Men strukturerede lasagneopskrifter, ala den vi viser i program 10.1 er almindelige i mange kogebøger.

I en mere struktureret opskrift indgår delopskrifter på lasagneplader, hvid sovs og kødsovs

```
Lav en dobbelt portion lasagneplader.

Lav en portion hvid sovs.

Lav en portion kødsovs.

Bland nu lagvis pastabaner, kødsovs og hvid sovs. Drys med paramesanost.

Gratiner retten i ovnen 15 minutter ved 225 grader.
```

Program 10.1 *Lasagne.*

De tre fremhævede linier aktiverer abstraktioner (delopskrifter) for hhv. det at lave lasagneplader (for vi køber dem nemlig ikke færdige i Bilka), hvid sovs, og kødsovs. Vi tænker på de sidste to trin som basale trin, der kan genkendes fra opskriften i afsnit 10.1.

Men vi skal naturligvis også beskrive, hvad det vil sige at lave plader, hvid sovs og kødsovs. Hver delopskrift *indkapsler* en række madlavningstrin. Vi viser de tre madlavningsabstraktioner i hhv. program 10.2, program 10.3 og program 10.4.

```
Bland og ælt 500 g durum hvedemel, 5 æg, lidt salt og 2 spsk olie;

Kør pastaen gemmen en pastamaskine i tynde baner;
```

Program 10.2 *Lav en dobbelt portion lasagneplader.*

```
Smelt 3 spsk smør, og rør det sammen med 3 spsk mel;

Spæd med med 5 mælk og kog sovsen under svag varme;
```

Program 10.3 *Lav en portion hvid sovs.*

```
Svits 2 hakkede løg med 500 g hakket oksekød,
samt salt og pebber;

Tilsæt 3 spsk tomatpuré, en ds. flåede tomater
og 3 fed knust hvidløg;

Kog kødsovsen 10 minutter;
```

Program 10.4 *Lav en portion kødsovs.*

Vi kunne forsøge at strække analogien lidt længere. For det første kunne vi forsøge os med en generalisering af en delopskrift til at dække forskellige retter med fælles kendetegn. En realistisk og brugbar generalisering kunne være omfanget af delretten (enkelt portion, dobbelt portion, etc.).

For det andet kunne vi forsøge os med opskrifter, som skulle bruge en given delopskrift mere end én gang. Dette er måske urealistisk i den gastronomiske verden, i det mindste relativ til denne forfatters begrænsede gastronomiske forestillingsevner.

Anvendelser af delopskrifter højner abstraktionsniveauet og muliggør genbrug af basale opskrifter.

Delopskrifter svarer til *procedurer* i programmeringssprog. Brug af parametre generaliserer en opskrift så den bliver mere alsidig.

10.3. Lasagne ala C

Lektion 3 - slide 4

Lad det være sagt klart og tydeligt allerede her - du får aldrig lasagne ud af din C compiler og fortolker. Alligevel forsøger vi os i program 10.5 med et C lignende program for den strukturerede lasagneopskrift, vi nåede frem til i afsnit 10.2.

I program 10.5 ser vi tre såkaldte funktionsprototyper lige efter include. Disse annoncerer at der er abstraktioner som hedder `make_lasagne_plates` `make_white_sauce`, og `make_meat_sauce`. Disse er nødvendige for at kunne anvende funktionerne i hovedopskriften. Så kommer *hovedprogrammet*, som altid i C hedder `main`. Efter denne ser vi implementeringen af de tre annoncerede funktioner.

At vi har forfattet program 10.5 på engelsk betyder naturligvis ikke alverden. Som regel foretrækker jeg at angive navne, kommentarer mv. i programmer på engelsk, for at opnå den bedste mulige helhed ift. selve programmeringssproget. Men det er udtryk for 'smag og behag'.

```
#include <stdio.h>

void make_lasagne_plates(void);
void make_white_sauce(void);
void make_meat_sauce(void)

int main(void) {
  make_lasagne_plates();
  make_white_sauce();
  make_meat_sauce();

  mix plates, meat sauce, and white sauce;
  sprinkle with paramesan cheese;
  bake 15 minutes at 225 degrees;

  return 0;
}

void make_lasagne_plates(void) {
  mix flour, egs, salt and oil;
  process the pasta in the pasta machine;
}

void make_white_sauce(void) {
  melt butter and stir in some flour;
  add milk and boil the sauce;
}
```

```
void make_meat_sauce(void){
  chop the onion, and add meat, salt and pebber;
  add tomatos and garlic;
  boil the sauce 10 minutes;
}
```

Program 10.5 *Et pseudo C program som laver lasagne.*

Det er hermed slut på det gastronomiske kvarter. I kapitel 11 fortsætter vi vores indledende udforskning af abstraktioner og funktioner i en meget simpel grafisk verden.

11. Tændstikgrafik abstraktioner

Tændstikgrafik anvendes her for simple stregtegninger af sammensatte figurer, f.eks. mennesker med hoved, arme, krop og ben.

11.1. Eksempel: En tændstikpige (1)

Lektion 3 - slide 6

I program 11.2 ser vi først en `main` funktion, som kalder en abstraktion, `prn_match_girl`. `prn_match_girl` udskriver konturerne af en pige i termer af hoved, arme, krop, ben ved at kalde `prn_head`, `prn_arms`, `prn_body` og `prn_legs`. Den ønskede tegning af pigen er vist herunder:

```
      *
    *   *
  *       *
  *       *
    *   *
      *
-------------
     /\
    /  \
   /    \
  /      \
 /        \
/          \
-------------
     /\
    /  \
   /    \
  /      \
 /        \
/          \
```

Program 11.1 *Det ønskede program output.*

Vi ønsker at tegne en 'tændstikpige' ved brug af simpel tegngrafik

```
int main(void){
  prn_match_girl();
  return 0;
}

void prn_match_girl(void){
  prn_head();
  prn_arms();
  prn_body();
  prn_legs();
}
```

Program 11.2 *Udskrivning af tændstikpige i termer af udskrivning af hoved, arme, krop og ben.*

Ovenstående er udtryk for en programmeringsidé som kaldes *programudvikling af trinvis forfinelse*. Vi anvender først `prn_match_girl` i `main`, uden at have denne til rådighed i forvejen. Når vi således har indset behovet for denne abstraktion, programmerer vi den som næste punkt på dagsordenen. Tilsvarende sker det med `prn_head`, `prn_arms`, `prn_body` og `prn_legs`.

Tegneprocessen udføres top-down. Det udtrykkes at en tændstikpige tegnes som et hoved, arme, krop og ben. I næste trin må vi definere, hvordan disse kropsdele tegnes.

11.2. Eksempel: En tændstikpige (2)

Lektion 3 - slide 7

Vi fortsætter den trinvise forfinelse af programmet som tegner tændstikpigen. Vi har brugt `prn_head`, `prn_arms`, `prn_body` og `prn_legs` i program 11.2, men vi har ikke lavet disse primitiver endnu. Det råder vi bod på i program 11.3.

Pigens hoved, arme, krop og ben tegnes ved kald af generelle geometriske tegne procedurer

```
void prn_head(void){
  prn_circle();
}

void prn_arms(void){
  prn_horizontal_line();
}

void prn_body(void){
  prn_reverse_v();
  prn_horizontal_line();
}

void prn_legs(void){
  prn_reverse_v();
}
```

Program 11.3 *Udskrivning af hoved, arme, krop og ben i termer af generelle geometriske figurer.*

Vi ser at et hoved tegnes som en cirkel, armene som en vandret streg, kroppen som et omvendt, lukket v, og benene som et omvendt v. Vi gør brug af tre nye primitiver, `prn_circle`, `prn_horizontal_line` og `prn_reverse_v`.

11.3. Eksempel: En tændstikpige (3)

Lektion 3 - slide 8

Der er nu tilbage at realisere tegningen af cirkler, vandrette streger og den omvendte v form. Dette gør vi på helt simpel vis i program 11.4.

Man kan spørge om vi virkelig ønsker så mange niveauer for den relativt simple overordnede tegneopgave, som vi startede i kapitel 11. Svaret er et klart ja.

Vi får for det første nedbrudt den komplekse og sammensatte tegneopgave i en række simple tegneopgaver. Generelt er det måden at beherske indviklede og komplicerede programmeringsudfordringer.

Genbrugelighed er en anden markant fordel. I tegningen af tændstikpigen bruges den omvendte V form to forskellige steder. Det er en helt åbenbar fordel at disse to steder kan deles om tegningen af denne form. Dette gøres ved at kalde `prn_reverse_v` fra både `prn_body` og `prn_legs`. Selve implementationen af `prn_reverse_v` findes naturligvis kun ét sted.

Cirkler, linier og andre former tegnes ved brug af primitiv tegngrafik

```c
#include <stdio.h>

void prn_circle(void){
  printf("      *          \n");
  printf("    *   *        \n");
  printf("  *       *      \n");
  printf("  *       *      \n");
  printf("    *   *        \n");
  printf("      *          \n");
}

void prn_horizontal_line(void){
  printf("------------    \n");
}

void prn_reverse_v(void){
  printf("     /\\         \n");
  printf("    /  \\        \n");
  printf("   /    \\       \n");
  printf("  /      \\      \n");
  printf(" /        \\     \n");
  printf("/          \\    \n");
}
```

Program 11.4 *Udskrivning af cirkler, linier og andre geometriske figurer.*

I program 11.4 ser det lidt underligt ud at der tilsyneladende udskrives en dobbelt backslash '\'. Dette skyldes at backslash er et såkaldt escape tegn, som påvirker betydningen af de efterfølgende tegn. Derfor noterer '\\' reelt et enkelt bagvendt skråstreg i en C tekststreng. Tilsvarende noterer '\n' et linieskift.

11.4. Eksempel: En tændstikpige (4)

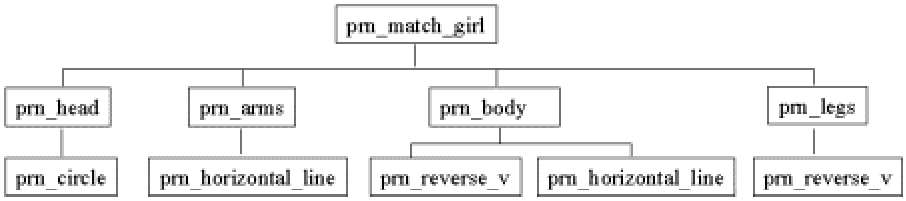
Lektion 3 - slide 9

I dette afsnit viser figur 11.1 et overblik over strukturen af de abstrationer, vi har indført i afsnit 11.1 - afsnit 11.3. Men først repeteres de essentielle observationer om den benyttede fremgangsmåde:

Programmet er lavet *top-down*.

Programmering ved *trinvis forfinelse*.

Programmet designes og implementeres i et antal niveauer med postulering og efterfølgende realisering af et antal procedurer



Figur 11.1 *En illustration of problemer og delproblemer i forbindelse med tegning af en tændstikpige*

Læg igen mærke til den hierarkiske nedbrydning og hvordan nogle abstraktioner bruges mere end ét sted.

11.5. Eksempel: En tændstikpige (5)

Lektion 3 - slide 10

I dette afsnit vil vi vise, hvordan vi konkret i C kan organisere de abstraktioner, som er præsenteret i de forrige afsnit. Som en central idé introducerer vi et separat *char graphics library*. I program 11.5 inkluderer vi dette bibliotek i den røde del. I den blå del skriver vi såkaldte funktionsprototyper for de funktioner, vi 'skubber foran os' i programmet. Dernæst følger abstraktionerne som printer hele pigen, hovedet, arme, krop og ben.

```
#include "char-graphics.h"

void prn_match_girl(void);
void prn_head(void);
void prn_arms(void);
void prn_body(void);
void prn_legs(void);

int main(void){
  prn_match_girl();
  return 0;
}

void prn_match_girl(void){
  prn_head();
  prn_arms();
  prn_body();
  prn_legs();
}

void prn_head(void){
  prn_circle();
}

void prn_arms(void){
  prn_horizontal_line();
}

void prn_body(void){
  prn_reverse_v();
  prn_horizontal_line();
```

```
}

void prn_legs(void){
  prn_reverse_v();
}
```

Program 11.5 *Tændstikpige programmet.*

I program 11.6 viser vi den såkaldte header fil for char graphics biblioteket. Det er denne fil vi inkluderede i den røde del af program 11.5.

```
/* Very simple graphical character-based library  */


/* Print a circle */
void prn_circle(void);

/* Print a horizontal line */
void prn_horizontal_line(void);

/* Print a reverse v shape */
void prn_reverse_v(void);
```

Program 11.6 *Filen char-graphics.h - header fil med prototyper af de grafiske funktioner.*

Herunder, i program 11.7 ser vi selve implementationen af det grafiske bibliotek. For hver funktionsprototype i program 11.6 har vi en fuldt implementeret funktion i program 11.7.

```
#include <stdio.h>

void prn_circle(void){
  printf("      *          \n");
  printf("    *   *        \n");
  printf("  *       *      \n");
  printf("  *       *      \n");
  printf("    *   *        \n");
  printf("      *          \n");
}

void prn_horizontal_line(void){
  printf("------------    \n");
}

void prn_reverse_v(void){
  printf("     /\\         \n");
  printf("    /  \\        \n");
  printf("   /    \\       \n");
  printf("  /      \\      \n");
  printf(" /        \\     \n");
  printf("/          \\    \n");
}
```

Program 11.7 *Filen char-graphics.c - implementationen af de grafiske funktioner.*

Endelig viser vi herunder hvordan vi først kan oversætte det grafiske bibliotek, og dernæst programmet som tegner tændstikpigen. Læg mærke til de forskellige compiler options, som giver udvidede fejlcheck.

```
* Compilation of the char-graphics.c library:

   gcc -ansi -pedantic -Wall -O -c char-graphics.c

* Compilation of the match-girl.c program:

   gcc -ansi -pedantic -Wall -O match-girl.c char-graphics.o
```

Program 11.8 *Oversættelse af programmerne - med mange warnings.*

Her følger nogle overordnede observationer af vores programorganisering:

> Funktioner skal erklæres før brug.
>
> Genbrugelige funktioner organiseres i separat oversatte filer.
>
> Prototyper af sådanne funktioner skrives i såkaldte header filer.

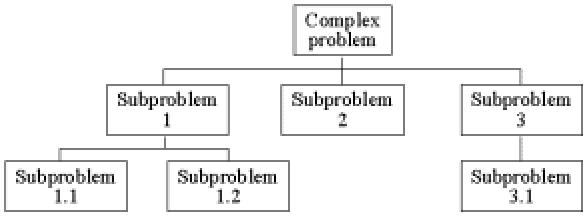
11.6. Del og hersk

Lektion 3 - slide 11

Efter at have overlevet det forholdsvis lange eksempel i de forrige afsnit er det nu tid til at gøre status.

Vi taler om *del og hersk problemløsning*. Ved at dele problemet op i mindre delproblemer hersker og behersker vi mange af de komplikationer, som vi støder på. Evnen til at dele problemer i mindre problemer er meget vigtig. Det er naturligvis tilfældet når vi prøver kræfter på store programmeringsopgaver. Men det gælder faktisk også ved løsning af de småopgaver, vi møder ved øvelserne i dette kursus. Jeg vil kraftigt opfordre alle til at tænke på dette næste gang I skriver et program!

> Komplekse problemer kan ofte opdeles i et antal simplere delproblemer.
>
> Delproblemernes løsninger kan ofte sammensættes til en løsning på det oprindelige problem.



Figur 11.2 *Opdelning af et kompleks problem i delproblemer*

- Del og hersk som top down programmering ved trinvis forfinelse:
  - Hvert delproblem P løses i en procedure
  - Procedurer, som er løsninger på delproblemer af P, placeres efter proceduren som løser P
    - Kræver erklæring af funktionsprototyper i starten af programmet
  - I kroppen af proceduren, som løser problemet P, programmeres en sammensætning af delproblemernes løsning

I næste kapitel vil vi se nærmere på funktionsbegrebet, herunder naturligvis funktioner i programmeringssproget C.

12. Procedurer og funktioner

Vi ser i dette kapitel på procedurer og funktioner generelt, og på funktionsbegrebet i C som dækker begge af disse.

12.1. Procedurer og funktioner

Lektion 3 - slide 13

Her følger kort vore definitioner af en procedure og en funktion:

En *procedure* er en abstraktion over en sekvens af kommandoer

En *funktion* er en abstraktion over et udtryk

Indkapsling er et helt basalt element i både procedurer og funktioner.

Et antal kommandoer sammensættes (aggregeres) i en procedure, og disse kan efterfølgende benyttes som én kommando i et procedurekald. Sammensætningen sker via en blok, som vi så på i kapitel 7.

I en funktion indkapsles ét udtryk, og indkapslingen spiller selv rollen som et udtryk, i et funktionskald.

- Procedure
  - En proceduredefinition indkapsler et antal kommandoer
  - Kaldet af en procedure er selv en kommando
- Funktion
  - En funktionsdefinition indkapsler netop ét udtryk
  - Kaldet af en funktion er selv et udtryk

Genbrugelighed og generalitet fremmes ved brug af parametre i funktioner. Vi ser konkrete eksempler på dette i afsnit 12.4 og de efterfølgende afsnit. Selve parametermekanismen diskuteres i afsnit 12.7.

12.2. Funktionsdefinitioner i C

Lektion 3 - slide 14

Både procedurer og funktioner, som defineret i afsnit 12.1 kaldes funktioner i C.

> C skelner ikke skarpt mellem procedurer og funktioner
>
> I C kaldes begge abstraktionsformer funktioner

Her følger den essentielle syntaks for en funktion i C:

```
type function_name (formal_parameter_list) {
  declarations
  commands
}
```

Syntaks 12.1 *En funktionsdefinition i C*

De *formelle parametre* er de parametre, som findes i selve funktionsdefinitionen. Når vi senere kalder funktionen angives *aktuelle parametre*, som erstatter de formelle parametre på en måde, som vi vil se nærmere på i forbindelse med eksemplerne i afsnit 12.4 - afsnit 12.7.

- Karakteristika:
  - Procedurer angiver *type* som **`void`**
  - Funktioner angiver *type* som en kendt datatype, f.eks. **`int`**, **`char`**, eller **`double`**
  - I procedurer og funktioner uden parametre angives *formal_parameter_list* som **`void`**

Mange C funktioner er reelt procedurer, som returnerer en eller anden værdi. Den returnerede værdi anvendes ofte til at signalere, om de indkapslede kommandoer 'gik godt' eller 'gik skidt'. I denne sammenhæng betyder returværdien 0 ofte, at alt er OK, medens en ikke-nulværdi som resultat er tegn på en eller anden fejl.

12.3. Funktionskald i C

Lektion 3 - slide 15

Når man kalder en funktion aktiveres kommandoerne i kroppen af funktionen på det vi kalder de aktuelle parametre. Et kald af funktionen står altså for de indkapslede kommandoer, hvor de aktuelle parametres værdier erstatter de formelle parametre. Dette kalder vi parameteroverførsel, og vi beskrive dette nærmere i afsnit 12.7.

```
function_name (actual_parameter_list)
```

Syntaks 12.2 *Et funktionskald i C*

- Karakteristika:
  - En funktion kan ikke kaldes før den er erklæret
  - Antallet af aktuelle parametre i kaldet skal modsvare antallet af formelle parametre i definitionen
  - En aktuel parameter er et udtryk, som beregnes inden den overføres og bindes til den tilsvarende formelle parameter

12.4. Eksempel: Temperaturomregninger

Lektion 3 - slide 16

Vi ser først på et eksempel på en ægte funktion. Hermed mener vi altså en funktion i den forstand vi definerede i afsnit 12.1, og som modsvarer det matematiske funktionsbegreb.

Funktionen `fahrenheit_temperature` tager som input en celcius temperatur, f.eks. 100 grader (kogepunktet ved normale forhold). Med dette output returnerer funktionen den tilsvarende Fahrenheit temperatur, 212 grader. Dette er den normale temperaturangivelse i Amerikanske lande, f.eks. USA.

Der udføres ingen kommandoer i C funktionen `fahrenheit_temperature`. Funktionen dækker med andre ord blot over en *formel*. Vi kalder ofte en formel for et *udtryk*, jf. afsnit 2.1.

```c
#include <stdio.h>

double fahrenheit_temperature(double celcius_temp){
  return (9.0 / 5.0) * celcius_temp + 32.0;
}

int main(void){

  printf("Freezing point: %6.2lf F.\n",
         fahrenheit_temperature(0.0));

  printf("Boiling point: %6.2lf F.\n",
         fahrenheit_temperature(100.0));

  return 0;
}
```

Program 12.1 *Et program som omregner frysepunkt og kogepunkt til Fahrenheit grader.*

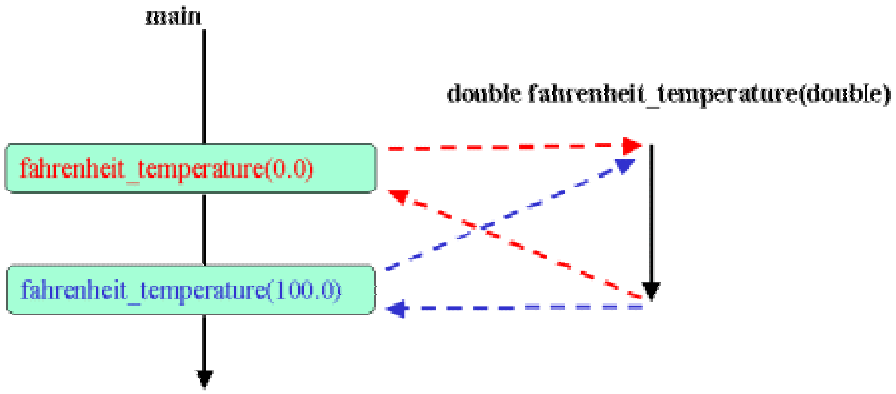
I programmet ser vi øverst, med fed sort skrift definitionen af `fahrenheit_temperature`. Bemærk udtrykket `(9.0 / 5.0) * celcius_temp + 32.0` , hvis værdi returneres fra funktionen med `return`. Bemærk også anvendelsen af navnet `celcius_temp`; Dette er den formelle parameter af funktionen.

I `main` ser vi to kald af `fahrenheit_temperature`. Dette er de røde og blå dele af program 12.1. Bemærk at disse dele er udtryk, og at de forekommer som parametre til `printf`.

Herunder, i figur 12.1 illustrerer vi det røde og blå kald af `fahrenheit_temperature` i program 12.1.

Kontrollen i `main` forløber som følger (følg pilene):

1. Indgang til `main` (lodret sort pil).
2. Det røde kald af `fahrenheit_temperature` (vandret rød pil til højre).
3. Første udførelse af `fahrenheit_temperature` (lodret sort pil).
4. Den røde returnering fra `fahrenheit_temperature` (opadrettet rød venstre pil).
5. Det blå kald af `fahrenheit_temperature` (vandret blå pil til højre).
6. Anden udførelse af `fahrenheit_temperature` (lodret sort pil).
7. Den blå returnering fra `fahrenheit_temperature` (blå pil mod venstre).
8. Udgang af `main` (lodret sort pil).



Figur 12.1 *To kald af fahrenheit_temperature funktionen*

Herunder, i program 12.2 viser vi et eksempel på brug af den omvendte funktion, som konverterer fahrenheit grader til celcius grader. Konkret programmerer vi en nyttig temperatur konverteringstabel. De første linier i udskriften er vist i program 12.3.

```
#include <stdio.h>

double celcius_temperature(double fahrenheit_temp){
  return (5.0 / 9.0) * (fahrenheit_temp - 32.0);
}

int main(void){
  double f;

  printf("%-20s %-20s\n", "Fahrenheit degrees", "Celcius degrees");

  for (f = 1.0; f <= 100.0; f++)
    printf("%-20.2lf %-20.2lf\n", f, celcius_temperature(f));

  return 0;
}
```

Program 12.2 *Et program som udskriver en nyttig temperaturkonverteringstabel.*

Bemærk kontrolstrengen i printf i program 12.2, såsom `%-20.2lf`. `lf` benyttes idet vi udskriver en double. `20.2` angiver antallet af cifre og antallet af cifre efter decimal point. Endelig angiver tegnet `-` and vi ønsker venstrestilling af output i feltet.

```
1.00                 -17.22
2.00                 -16.67
3.00                 -16.11
4.00                 -15.56
5.00                 -15.00
6.00                 -14.44
7.00                 -13.89
8.00                 -13.33
9.00                 -12.78
...                  ...
```

Program 12.3 *Partiel output fra ovenstående program.*

12.5. Eksempel: Antal dage i en måned

Lektion 3 - slide 17

Eksemplet i dette afsnit er en fortsættelse af eksemplet fra program 8.4. Bidraget i dette afsnit er en pæn indpakning af de relevante programdele i en funktion, og en lettere omskrivning, som går på at vi nu tager udgangspunkt i et månedsnummer og et årstal.

Bemærk lige kontrasten mellem program 8.4 og program 12.4. I det første program var vi i stand til at beregne antallet af dage i en måned som en del af `main`. I det sidste program har vi en veldefineret byggeklods, funktionen `daysInMonth`, som kan bruges så ofte vi vil. Dette er en meget mere modulær løsning. En løsning der på alle måder er mere tilfredsstillende.

Vi introducerer en funktion i programmet der beregner antal dage i en måned Vi bruger eksemplet til at diskutere overførsel af parametre til funktioner

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

int daysInMonth(int mth, int yr);
int isLeapYear(int yr);

int main(void) {

  int mth, yr;

  do{
    printf("Enter a month - a number between 1 and 12: ");
    scanf("%d", &mth);
    printf("Enter a year: ");
    scanf("%d", &yr);

    printf("There are %d days in month %d in year %d\n",
              daysInMonth(mth, yr), mth, yr);
  } while (yr != 0);

  return 0;
}

int daysInMonth(int month, int year){
  int numberOfDays;
  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 (isLeapYear(year)) numberOfDays = 29;
      else numberOfDays = 28; break;
    default: exit(-1);  break;
  }
```

```
  return numberOfDays;
}

int isLeapYear(int year){
  int res;
  if (year % 400 == 0)
    res = 1;
  else if (year % 100 == 0)
    res = 0;
  else if (year % 4 == 0)
    res = 1;
  else res = 0;
  return res;
}
```

Program 12.4 *Et program der beregner antal dage i en måned med en funktion.*

I program 12.4 bemærker vi først, at vi har prototyper af funktionerne `daysInMonth` og `isLeapYear` (fremhævet med fed sort skrift). Dernæst følger `main`, som kalder `daysInMonth` (den blå del). Endelig følger definitionen af `daysInMonth` med rød skrift. Som input tager funktionen parametrene `month` og `year`, og som output returneres antallet af dage i den ønskede måned. Læg mærke til denne meget veldefinerede grænseflade mellem funktionen `daysInMonth` og dets omverden.

Vi har øget genbrugeligheden - og dermed 'værdien' af vort program

Vi ønsker at undgå brug af `scanf` og `printf` i genbrugelige procedurer og funktioner

12.6. Eksempel: GCD

Lektion 3 - slide 18

Vi vender tilbage til et andet eksempel fra kapitel 9, nemlig Euclids algoritme, som vi oprindelig mødte i program 9.1. Vi husker at Euclids algoritme finder den største fælles divisor af to tal. Altså det største tal, som både går op i det ene og det andet tal.

Mønstret fra dage-i-måned eksemplet, program 12.4, genfindes klart i program 12.5. Læg mærke til funktionen `gcd`'s grænseflader. På inputsiden er der helt naturligt to heltal, hvoraf vi ønsker at finde den største fælles divisor. På outputsiden resultatet, altså den største fælles divisor.

```c
#include <stdio.h>

int gcd(int, int);

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

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

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

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

  return 0;
}

int gcd(int large, int small){
  int remainder;
  while (small > 0){
    remainder = large % small;
    large = small;
    small = remainder;
  }
  return large;
}
```

Program 12.5 *Et program der beregner største fælles divisor med en funktion.*

12.7. Parametre til C funktioner

Lektion 3 - slide 19

Som afslutning på dette afsnit ser vi lidt mere systematisk på parametre til funktioner i C.

Vi har mødt eksempler på parametre i både program 12.1, program 12.4 og program 12.5. I dette afsnit ser vi på den faktiske mekanisme til parameteroverførsel i C. Dette er de såkaldte værdiparametre, som på engelsk ofte betegnes som "call by value".

Parametre til en funktion bruges til at gøre funktionen mere generelt anvendelig

En parameter, som optræder i en funktionsdefinition, kaldes en *formel parameter*. En formel parameter er et navn.

En parameter, som optræder i et kald af en funktion, kaldes en *aktuel parameter*. En aktuel parameter er et udtryk, som beregnes inden det overføres.

- Regler for parameteroverførsel:
  - Antallet af aktuelle parametre i kaldet skal modsvare antallet af formelle parametre i definitionen
  - Typen af en aktuel parameter skal modsvare den angivne type af hver formel parameter
    - Dog muligheder for visse implicitte typekonverteringer
  - Parametre til funktioner i C overføres som *værdiparametre*
    - Der overføres en kopi af den aktuelle parameters værdi
    - Kopien bindes til (assignes til) det formelle parameternavn
    - Når funktionen returnerer ophører eksistensen af de formelle parametre, på samme måde som lokale variable i funktionen.

13. Eksempel: Rodsøgning

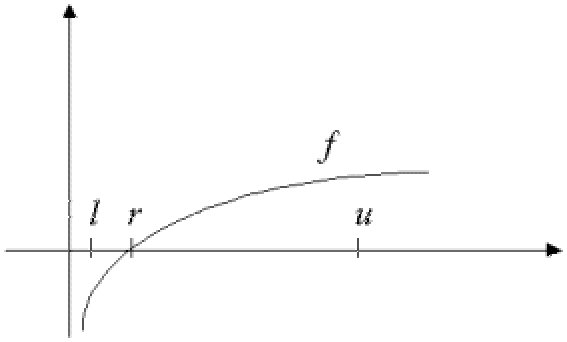
Vi gennemgår her et lidt mere avanceret eksempel, nemlig rodsøgning i en kontinuert funktion. Vores primære interesse er naturligvis problemløsning og funktioner, herunder topdown udvikling og programmering ved trinvis forfinelse.

13.1. Rodsøgning (1)

Lektion 3 - slide 21

Vi starter med den essentielle observation, der danner baggrund for den centrale rodsøgningsalgoritme:

Enhver kontinuert funktion *f*, som har en negativ værdi i *l* og en positiv værdi i *u* (eller omvendt) vil have mindst én rod *r* mellem *l* og *u*.



Figur 13.1 *En kontinuert funktion f med en rod mellem l og u*

I figur 13.1 ser vi tydeligt at der skal være mindst én rod mellem *l* og *u*, nemlig i *r*. Principielt kan der være flere. Hvis dette er tilfældet har vi ingen egentlig kontrol over, hvilken én vi finder.

Ved at indsnævre intervallet [*l*, *u*], og ved hele tiden at holde roden mellem *l* og *u*, kan vi hurtigt finde frem til en værdi *r* for hvilken *f(r) = 0*.

13.2. Rodsøgning (2)

Lektion 3 - slide 22

Den central algoritme, som finder en rod mellem *l* og *u* er vist i program 13.1. Bemærk, at vi forudsætter at *l* er mindre end (eller lig med) *u*, og at fortegnet af funktionens værdi i *l* og *u* er forskellige. Med andre ord skal situationen være som i figur figur 13.1.

```
double findRootBetween(double l, double u){
 while (!isSmallNumber(f(middleOf(l,u)))){
   if(sameSign(f(middleOf(l,u)), f(u)))
     u = middleOf(l,u);
   else
     l = middleOf(l,u);
 }
 return middleOf(l,u);
}
```

Program 13.1 *Rodsøgningsfunktionen.*

I programmet flytter vi gentagne gange enten `l` eller `u` til midtpunktet . Bemærk her, at `l` og `u` er parametrene i funktionen `findRootBetween`, og når vi ændrer på `l` eller `u` er det kun den lokale værdi i funktionen der ændres. Dette er en konsekvens af brug af værdiparametre (se afsnit 12.7).

De røde, blå og lilla dele af `findRootBetween` i program 13.1 skal nu programmeres. I alle tre tilfælde er der tale om småfunktioner, der medvirker til at højne abstraktionsniveauet i `findRootBetween`. I program 13.2 viser vi mulige definitionen af disse funktioner.

```
int sameSign(double x, double y){
  return x * y >= 0.0;
}

double middleOf(double x, double y){
  return x + (y - x)/2;
}

int isSmallNumber(double x){
  return (fabs(x) < 0.0000001);
}
```

Program 13.2 *Funktionerne sameSign, middleOf og isSmallNumber.*

Vi kan nu samle alle dele i ét program. Alternativt kunne vi have organiseret 'stumperne' i to eller flere kildefiler, ligesom vi gjorde i tændstikpige eksemplet i afsnit 11.5.

Vi bemærker, at vi i `main` programmerer en løkke, som gentagne gange beder brugeren af programmet om intervaller, hvor vi søger efter rødder i funktionen $x^3 - x^2 - 41\,x + 105$. Som antydet ved vi at der er rødder i 5.0, 3.0 og -7.0. Det skyldes at $x^3 - x^2 - 41\,x + 105$ faktisk bare er lig med $(x - 5.0) \cdot (x - 3.0) \cdot (x + 7.0)$. Prøvekør selv programmet, og find rødderne!

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

double f (double x){
  /* (x - 5.0) * (x - 3.0) * (x + 7.0) */
  return (x*x*x - x*x - 41.0 * x + 105.0);
}

int sameSign(double x, double y){
  return x * y >= 0.0;
}

double middleOf(double x, double y){
  return x + (y - x)/2;
}

int isSmallNumber(double x){
  return (fabs(x) < 0.0000001);
}

double findRootBetween(double l, double u){
 while (!isSmallNumber(f(middleOf(l,u)))){
   if(sameSign(f(middleOf(l,u)), f(u)))
     u = middleOf(l,u);
   else
     l = middleOf(l,u);
 }
 return middleOf(l,u);
}

int main (void){
    double x, y;
    int numbers;

    do{
      printf("%s","Find a ROOT between which numbers: ");
      numbers = scanf("%lf%lf", &x, &y);

      if (numbers == 2 && !sameSign(f(x),f(y))){
          double solution = findRootBetween(x,y);
          printf("\nThere is a root in %lf\n", solution);
        }
      else if (numbers == 2 && sameSign(f(x),f(y)))
          printf("\nf must have different signs in %lf and %lf\n",
                  x, y);
      else if (numbers != 2)
          printf("\nBye\n\n");
    }
    while (numbers == 2);
}
```

Program 13.3 *Hele rodsøgningsprogrammet.*

14. Rekursive funktioner

En rekursiv funktion er kendetegnet ved, at den kalder sig selv. Men som det mest væsentlige, er rekursive funktioner udtryk for en problemløsning, hvor delproblemerne ligner det overordnede problem.

Vi introducerer rekursion i dette kapitel. I en senere lektion, startende i kapitel 32, vender vi tilbage til rekursion med fornyet styrke.

14.1. Rekursive funktioner

Lektion 3 - slide 24

Først gør vi os de essentielle iagttagelser om rekursion, problemer som har rekursivt forekommende delproblemer:

> En rekursiv funktion kalder sig selv
>
> Rekursive funktioner er nyttige når et problem kan opdeles i delproblemer, hvoraf nogle har samme natur som problemet selv

Som det første eksempel programmerer vi fakultetsfunktionen. Hvis vi af bekvemmelighed her kalder fakultetsfunktionen for *f*, gælder at $f(n) = n \cdot (n-1) \cdot (n-2) \cdot \ldots 2 \cdot 1$. Dette kan også udtrykkes rekursivt som følger:

- $f(n) = n \cdot f(n-1)$ for $n > 0$
- $f(0) = 1$

Denne rekursive formel er udtrykt direkte i C funktionen `factorial` i program 14.1. Det rekursive kald er fremhævet med rødt.

```
#include <stdio.h>

unsigned long factorial(unsigned long n){
  if (n == 0)
    return 1;
  else return n * factorial(n - 1);
}

int main(void) {
  unsigned long k;

  for (k = 1; k <= 12; k++)
    printf("%-20lu %20lu\n", k, factorial(k));

  return 0;
}
```

Program 14.1 *Et program med en rekursivt defineret fakultetsfunktion.*

Som de fleste nok ved i forvejen vokser fakultetsfunktionen ganske voldsomt. Så voldsomt at vi højst kan beregne `factorial(12)`, selv med tal i typen `long`. Vi viser outputtet af program 14.1 i program 14.2.

```
1                                       1
2                                       2
3                                       6
4                                      24
5                                     120
6                                     720
7                                    5040
8                                   40320
9                                  362880
10                                3628800
11                               39916800
12                              479001600
```

Program 14.2 *Output fra ovenstående program.*

Som endnu et eksempel viser vi en rekursiv udgave af rodsøgningsfunktionen `findRootBetween` i program 14.3. Vi mødte den iterative (gentagende) udgave i program 13.1. Gentagelsen blev i dette program lavet med en while løkke.

```
double findRootBetween(double l, double u){
  if (isSmallNumber(f(middleOf(l,u))))
     return middleOf(l,u);
  else if (sameSign(f(middleOf(l,u)), f(u)))
     return findRootBetween(l, middleOf(l,u));
  else if (!sameSign(f(middleOf(l,u)), f(u)))
     return findRootBetween(middleOf(l,u),u);
  else exit (-1);
}
```

Program 14.3 *En rekursiv version af funktionen findRootBetween.*

Vi lægger mærke til, at der ikke anvendes nogen form for løkker i `findRootBetween` som programmeret i program 14.3. Gentagelsen bestyres af at funktionen kalder sig selv et tilstrækkeligt antal gange, indtil vi er tæt nok på roden.

15. Korrekthed af programmer

Vi slutter denne lektion af med et kapitel om programkorrekthed.

15.1. Assertions

Lektion 3 - slide 26

Ved brug af logiske udtryk kan vi ofte få et program til at checke sig selv. Vi kalder ofte sådanne udtryk for *assertions*. Hvis et assertion beregnes til *false* (altså 0 i C) stoppes programmet med en informativ besked om, hvad der er galt.

Det er muligt at 'dekorere' et program med logiske udtryk (assertions), som fortæller os om programmet opfører sig som vi forventer

- To specielt interessante assertions:
  - **Præbetingelse af en funktion**: Fortæller om det giver mening at kalde funktionen
  - **Postbetingelse af en funktion**: Fortæller om funktionen returnerer et korrekt resultat

Vi illustrerer ideen om assertions med en forkert og en korrekt implementation af en kvadratrodsfunktion. Først, i program 15.2, en underlig og meget forkert implementation af my_sqrt som blot returnerer konstanten 7.3 uanset input.

Den røde prebetingelse i program 15.1 udtrykker, at parameteren til `my_sqrt` skal være ikke-negativ. Den blå postbetingelse udtrykker, at kvadratet af resultatet, altså res^2, skal være meget tæt på det oprindelige input `x`. Vi kan ikke forvente at ramme `x` eksakt, idet der jo altid vil være risiko for afrundingsfejl når vi arbejder med reelle tal på en computer.

Med den viste implementation af `my_sqrt` vil postbetingelsen (den blå assertions) fejle for de fleste input.

```
#include <stdio.h>
#include <math.h>
/* #define NDEBUG 1 */
#include <assert.h>

int isSmallNumber(double x){
  return (fabs(x) < 0.0000001);
}

double my_sqrt(double x){
  double res;
  assert(x >= 0);
  res = 7.3;
  assert(isSmallNumber(res*res - x));
  return res;
}

int main(void) {

  printf("my_sqrt(15.0): %lf\n", my_sqrt(15.0));
  printf("my_sqrt(20.0): %lf\n", my_sqrt(20.0));
  printf("my_sqrt(2.0): %lf\n", my_sqrt(2.0));
  printf("my_sqrt(16.0): %lf\n", my_sqrt(16.0));
  printf("my_sqrt(-3.0): %lf\n", my_sqrt(-3.0));

  return 0;
}
```

Program 15.1 *En forkert implementation af my_sqrt.*

Den udkommenterede konstant `NDEBUG` kan bruges til at styre, om assertions rent faktisk efterchecks. Hvis kommentarerne fjernes, vil assertions ikke blive beregnet.

Lad os også vise en korrekt implementation af en kvadratrodsfunktion. Vi benytter rodsøgningsprogrammet fra afsnit 13.2 til formålet. Den centrale observation er, at vi kan finde kvadratroden af k ved at finde en rod i funktionen $f(x) = x^2 - k$ i intervallet 0 og k. Det er nøjagtigt hvad vi gør i program 15.2 .

Bemærk de røde og blå dele i program 15.2, som er hhv. prebetingelse og postbetingelse af `my_sqrt`, ganske som i den forkerte udgave af kvadratrodsfunktionen i program 15.1. Den lilla del er funktionen `f`, som vi diskuterede ovenfor.

```
#include <stdio.h>
#include <math.h>
/* #define NDEBUG 1 */
#include <assert.h>

int isSmallNumber(double x){
  return (fabs(x) < 0.0000001);
}

double displacement;

double f (double x){
  return (x * x - displacement);
}

int sameSign(double x, double y){
  return x * y > 0.0;
}

double middleOf(double x, double y){
  return x + (y - x)/2;
}

double findRootBetween(double l, double u){
 while (!isSmallNumber(f(middleOf(l,u)))){
   if(sameSign(f(middleOf(l,u)), f(u)))
     u = middleOf(l,u);
   else
     l = middleOf(l,u);
 }
 return middleOf(l,u);
}


double my_sqrt(double x){
  double res;
  assert(x >= 0);
  displacement = x;
  res = findRootBetween(0,x);
  assert(isSmallNumber(res*res - x));
  return res;
}

int main(void) {
```

```
  printf("my_sqrt(15.0): %lf\n", my_sqrt(15.0));
  printf("my_sqrt(20.0): %lf\n", my_sqrt(20.0));
  printf("my_sqrt(2.0): %lf\n", my_sqrt(2.0));
  printf("my_sqrt(16.0): %lf\n", my_sqrt(16.0));

  return 0;
}
```

Program 15.2 *En brugbar implementation af my_sqrt implementeret via rodsøgningsfunktionen.*

Vi kan også illustrere assertions i selve rodsøgningsfunktionen `findRootBetween`, som oprindelig blev vist i program 13.1. Dette er illustreret i program 15.3 herunder.

I den røde prebetingelse af `findRootBetween` udtrykker vi at `l` skal være mindre end eller lige med `u` og at fortegnet af `f(l)` og `f(u)` skal være forskellige. I den blå postbetingelse kræver vi at `f(res)` er meget tæt på 0. Dette udtrykker at funktionen `findRootBetween` er korrekt på de givne input.

```
double findRootBetween(double l, double u){
 double res;
 assert(l <= u); assert(!sameSign(f(l), f(u)));
 while (!isSmallNumber(f(middleOf(l,u)))){
   if(sameSign(f(middleOf(l,u)), f(u)))
     u = middleOf(l,u);
   else
     l = middleOf(l,u);
 }
 res = middleOf(l,u);
 assert(isSmallNumber(f(res)));
 return res;
}
```

Program 15.3 *En udgave af findRootBetween med præbetingelse og postbetingelse.*