Afsnit 1

Med denne tutorial vil du kunne lære at programmere i C++, du kan af naturlige grunde ikke lære alt om C++ i en så kort artikel, men nok til at komme i gang. De programmer du vil lære er standard konsol programmer, dels fordi det er det letteste og dels fordi der ikke er en standard måde at lave GUI programmer på.

Artiklen er skrevet til brugere der ikke har programmeret før så alle kan være med. Jeg vil foreslå at du læser artiklen lidt af gangen, og prøver at løse nogle af de opgaver der bliver stillet undervejs, og måske prøver at lege lidt med det du har lært. På den måde tror jeg at du lærer mest og det er lang sjovere at lege lidt ind imellem. Hvis du har programmeret før vil du nok finde de første dele lidt kedelige, men læs dem alligevel.

Først lidt om sproget. C++ er et af verdens mest brugte programmeringssprog, det kan bruges til at lave stort set alle typer applikationer, fra softwaren i din MP3 afspiller over styresystemer, som f.eks. windows til applikationer som en internet browser. Sproget bruges til at programmere applikationer, men ikke til at lave web-sider med. C++ kode kan oversættes til stor set alle typer hardware.

Nå nok snak, lad os se på lidt kode:

```
#include <iostream>

int main()

{

   std::cout << "Hello World" << std::endl;

   std::cin.get();

}
```

Hvis du synes at det ligner græsk så fortvivl ikke, inden artiklen er omme vil du synes at det er logik for begyndere.

Lad os tage programmet fra en ende af og se hvad der sker.
Først

```
#include <iostream>
```

Dette betyder at vi ”inkluderer” filen iostream i vores program. Det er nødvendigt for at kunne bruge de objecter vi senere bruger til at skrive til skærmen og læse fra tastaturet. Man siger at iostream er en header-fil.

Alle programmer indeholder en main:

```
int main()

{

}
```

main er der hvor programmet starter. main er en funktion, og har derfor som alle andre funktioner i C++ () efter navnet.
{ og } angiver starten og enden på selve koden for main, man siger at {} bliver brugt til at angive en blok, programmet vil køre fra { til } og udføre en linie eller rettere statement af gangen.
int er typen på det vi giver tilbage til dem der kaldte vores program, vi siger at det er det vi

returnerer. Det er et tal, 0 betyder normalt at alt gik godt. Da vores program ikke kan fejle, angiver vi ikke hvad vi vil returnere, så systemet sørger for at der bliver returneret 0.

Nu til den del der faktisk laver noget, først denne:

```
std::cout << "Hello World" << std::endl;
```

Der er faktisk hele 6 elementer i den linie, så lad os tage dem en af gangen.
Først std::cout. cout er et object der repræsenterer skærmen, vi kan skrive på skærmen ved at sende ting til den. cout er en forkortelse af console-output.
<< er en operator der sender ting til en output "stream". Dvs den sender det der står til højre over til det der står til venstre.
"Hello World" er en streng, dvs en række af tegn. I vore eksempel er det det der bliver skrevet til skærmen. En streng starter med en " og ender med en ", disse udskrive ikke.
Den anden << laver det samme som den første. Det viser at vi kan sende alt det til skærmen på én gang som vi ønsker.
std::endl er et "linieskift", dvs at den bevirker, når vi sender den til cout at cursoren bliver flytte til starten på næste linie.
; markerer at vi er nået til enden på et "statement". Alle statements i C++ ender på en ;
Alt i alt betyder linjen at vi skriver Hello World på skærmen og flytter cursoren til starten på næste linie.

Sidste linie i koden

```
std::cin.get();
```

Læser et tegn fra tastaturet, og venter på at du trykker Enter, eller blot på at du trykker Enter. Det der er læst kastes bort. Dette er ikke strengt nødvendigt, men gør at hvis vi kører programmet fra explorer eller lign, vil det ikke lukke ned inden du får læst hvad det skrev.
Det kan være en god idé at tilføje denne linie til alle de andre programmer i denne guide, jeg har undladt den for at spare plads.
Det skal bemærkes at der i C++ er forskel på store og små bogstaver, så Cout er ikke det samme som cout.
std:: foran cout, cin osv. bruges fordi de ligger i standard namespace. Hvis ikke du gider skrive det igen og igen, kan du bare skrive denne linie efter dine #include:

```
using namespace std;
```

I denne tutorial vil jeg altid skrive std:: foran, for at du kan se at det er noget vi har hentet fra std.

Afsnit 2

Nu sidder du jo nok og tænker, ja ja, det er meget godt, men hvordan får jeg det til at køre, jeg vil gerne se noget ske i virkeligheden.

Så lad os komme til sagen. For at kunne køre koden har du brug for to ting. Et program til at skrive koden i, så du kan gemme det som en fil, dette kaldes en editor. Den anden ting du skal bruge er en compiler til at oversætte koden til et program.

Hvis ikke du allerede har bestemt dig for hvad du vil bruge vil jeg anbefale Code::Blocks. Det er en gratis compiler og editor i et. Det gør det let at arbejde med koden, oversætte og køre programmet. Du finder den her:

http://www.codeblocks.org/

Husk at downloade versionen med MinGW compileren.

Code::blocks kan kan også downloades i en version til Linux. På Linux er selve compileren

sandsynligvis installeret sammen med OS'et.

Når du har downloadet og installeret Code::Blocks skal du starte med at lave et nyt project:

Menu->File->New Project, så vælger du ”Console Application” og C++ source.

Den vil så selv oprette en main.cpp fil, der ligner eksemplet fra før meget, du kan selv sætte resten ind. Du finder main.cpp i boxen til venstre.

Så trykker du på F9, og den vil oversætte og køre koden. Efter lidt tid skulle der gerne dukke en konsol box op med teksten Hello World, og du har lavet, oversat og kørt dit første C++ program!

Hvis dette virker så prøv at lege lidt med koden.

Når du kører programmet fra Code::Blocks behøver du ikke std::cin.get() til slut, Code::Blocks sørger selv for at programmet ikke lukker før du beder om det.

Opgaver:

Prøv at udskrive mere tekst.

Prøv at lave en fejl i koden og se hvad der sker, prøv f.eks. at fjerne et semikolon eller indsætte nogle flere.

Afsnit 3

Nu er der jo grænser for hvor sjovt det er at skrive tekst ud på skærmen, så lad os prøve at læse noget fra tastaturet og skrive det ud igen.

For at læse har vi brug for noget til at opbevare det der er læst indtil vi skriver det ud igen, dette noget kunne man kalde en beholder, eller en variabel. En variabel har et navn, for at kunne skelne de forskellige variabler. I C++ har variabler også en type, det er typen der bestemmer hvad der kan opbevares i variablen, typen kunne f.eks. angive at variablen bliver brugt til tal, eller bogstaver. I C++ skal man fortælle compileren at variable findes, inden de kan bruges.

Vi vil nu lave et lille program der spørger brugeren om hvad han hedder, og skriver Hej navn. Den mest velegnede type til at opbevare et navn er en std::string. Så lad os oprette en variabel:

```
std::string DitNavn;
```

Her er ses det at typen er std::string, navnet på variablen er DitNavn. Man kan kalde sine variable stort set alt, man kan bruge bogstaver, dog ikke æøå, tal og _ til at lave navnet. Det første tegn må dog ikke være et tal. Det er ikke nogen god idé at starte navnet med _. Prøv at finde gode sigende navne på din variable, det hjælper med til at huske hvad de bliver brugt til.

Til at læse navnet fra tastaturet er std::getline velegnet, så programmet kunne se sådan ud:

```
#include <iostream>
#include <string>
int main()
{
   std::cout << "Hvad hedder du: " << std::endl;
   std::string DitNavn;
   std::getline(std::cin, DitNavn);
   std::cout << "Hej " << DitNavn << std::endl;
   std::cin.get();
}
```

Vi inkluderer string for at kunne bruge typen std::string og getline.

Linien med std::getline læser venter på at du skriver dit navn og trykker Enter.

Den sidste linie med std::cout skriver først Hej og derefter det du har skrevet at du hedder.

Opgave:

Prøv at skrive koden ind, køre programme og se hvad der sker.

Prøv også at lave flere variabler, og læse flere ting ind, f.eks. din adresse.

Nu vil vi prøve at lave et program der beder om to heltal, skriver summen, differencen, osv af de to tal.

Typen for et heltal i C++ er normalt int, vi har brug for to af dem, så:

int x, y;

Til at læse tal fra tastaturet kan man bruge >> operatoren:

std::cin >> x;

Dermed kan programmet komme til at se sådan ud:

```
#include <iostream>
int main()
{
   int x, y;
   std::cout << "Tal 1: ";
   std::cin >> x;
   std::cout << "Tal 2: ";
   std::cin >> y;
   std::cout << x << " + " << y << " = " << x + y << std::endl;
}
```

Som det ses læses de to tal, og vi skriver noget i stil med:

2 + 3 = 5

Opgave:

Udbyg programmet så det skriver x – y, x*y og x/y

Hvad sker der hvis y = 0, dvs. du dividerer med 0

Hvor store tal kan programmet regne med?

Hvad sker der hvis du skriver dit navn i stedet for det første tal?

Hvad mener programmet at 11 / 4 er?

Afsnit 4

Som du så ovenfor var det ikke så smart at forsøge at dividere med 0, så det var måske en idé at checke om tallet var 0 inden vi forsøger at dividere. Dvs. vi skal lave en betingelse, den laver man med if:

```
if(y != 0)
{
   std::cout << x << " / " << y << " = " << x / y << std::endl;
```

```
}
```

Det der står i () efter if er betingelsen, det der står inden for {} blokken udføres kun hvis betingelsen er sand. != betyder ”forskellig fra” eller ”ikke lig med”. Så det der står i {} udføres kun hvis y ikke er 0.

Det er ikke strengt nødvendigt at have {} omkring std::cout << ...., fordi der kun er ét statement i blokken, så vi kunne lige så godt skrive:

```
if(y != 0)
   std::cout << x << " / " << y << " = " << x / y << std::endl;
```

Hvad man foretrækker er mest et spørgsmål om smag og behag.

I vores kode var det måske en god idé at fortælle brugeren at y var 0 og vi derfor ikke kunne dividere hvis y var 0. Vi kunne skrive dette som

```
if(y != 0)
{
   std::cout << x << " / " << y << " = " << x / y << std::endl;
}
else
{
   std::cout << "Tal 2 er 0" << std::endl;
}
```

Her vil blokken efter else blive udført hvis y var 0.

Til at sammenligne kan man bl.a. bruge følgende:

== Det samme

!= Ikke det samme

> Større end

< Mindre end

>= Større end eller det samme

<= Mindre end eller det samme

&& Og

|| Eller

Man kan bygge disse sammen, hvis man f.eks. vil finde ud af om x er mindre end 10 eller større end 100 kan man skrive:

```
if(x < 10 || x > 100)
{
   std::cout << "x er ..." << std::endl;
}
```

Man kunne også finde ud af om x var større end eller lig med 10 og mindre end eller lig med 100:

```
if(x >= 10 && x <= 100)
{
   std::cout << "x er ..." << std::endl;
}
```

Man kan bygge flere if else sammen:

```
if(x < 10)
   std::cout << "x er mindre end 10" << std::endl;
else if(x < 20)
   std::cout << "x er større end 10 men mindre end 20" << std::endl;
else
   std::cout << "x er større end eller lig med 20" << std::endl;
```

Man kan også lave:

```
if(x > 10)
{
   if(x < 20)
      std::cout << ”Opgave 1” << std::endl;
   else
      std::cout << ”Opgave 1” << std::endl;
}
else
{
      std::cout << ”Opgave 1” << std::endl;
}
```

Og nu til opgaver:

1: Find ud af hvad der skal stå de tre steder i stedet for ”Opgave1” i eksemplet ovenfor. Kør derpå koden for at se om det er rigtigt. Brug koden fra afsnit 3 som en start.

2: Med koden fra afsnit 3 checker du om både x og y er større end 100, hvis de er udskriver du x/y ellers skriver du det største af de to divideret med det andet.

Afsnit 5

I dette afsnit skal vi kikke lidt mere på at regne, fordi det faktisk er det computeren gør det meste af tiden, og fordi det forekommer ofte i programmer har man i C++ lavet en lang række ting for at gøre det nemmere for en.

Vi har allerede lært om plus, minus, gange og dividere.

Den næste der er rar at kende er modulus operatoren. Hvis du har:

```
x = y % z;
```

Så bliver x resten i divisionen af y / z. Hvis y er 11 og z er 4 vil 11/4 være 2, og resten er 3, dvs. x bliver 3 i vores eksempel. Det er lidt hvis ikke man havde % operatoren måtte vi skrive:

```
x = y - (y / z)*z;
```

Og det er jo knap så elegant. Hvis y % z er 0 siger vi at z går op i y. Ofte bruger man også y %2 til at finde ud af om y er lige, hvis y % 2 er 0 er y lige, ellers er det ulige.

Tit har man brug for at lægge et tal til en variabel, så kan man skrive:

```
x = x + 3
```

Men da vi er dovne kan man skrive:

```
x += 3;
```

i stedet . Det samme gælder for +-*/.

Ofte har man også brug for at tælle en variabel 1 op eller ned, det kunne man gøre med:

```
x += 1;
```

Men igen er vi dovne, og kan bare skrive:

```
x++;
```

Det er så ikke så svært at regne ud hvorfor sproget hedder C++ ;-)

Der er to udgaver af ++ og --:

```
x++;
```

```
++x;
```

Hvis de står alene som her gør de det samme, men hvis vi skriver:

```
int x = 1;
```

```
int y = x++;
```

Vil det give noget andet end

```
int x = 1;
```

```
int y = ++x;
```

I det første eksempel vil y blive 1 men i andet eksemel vil y blive 2. Det skyldes at når man skriver ++x tælles x op inden man læser resultatet, ved x++ tælles x op efter at man har læst resultatet.

Hvis man har store regnestykker kan man bruge parenteser () til at bestemme hvad der skal regnes først, det der står inde i parentesen vil blive udregnet først.

Opgaver:

1: Find ud af om der er forskel på disse to, hvori består forskellen:

```
x + y * 2
```

```
(x + y) * 2
```

2: Jeg skrev tidligere at disse to giver det samme:

```
x = y % z;
```

```
x = y - (y / z)*z;
```

Find ud af om det er rigtigt.

Afsnit 6

Den koden vi har lavet indtil nu har kørt én gang fra start til slut. Med if/else kunne vi bestemme at visse dele skulle eller skulle ikke køres. Vi skal nu se på hvordan man får programmet til at gøre det samme flere gange efter hinanden. For at kunne det skal programmet springe tilbage, og køre det samme kode igen. Vi siger at programmet kommer til at køre i en løkke eller loop.

Først kikker vi på en for-loop:

```
#include <iostream>
int main()
{
   int counter;
   for(counter  = 0; counter < 4; counter++)
   {
      std::cout << "Hej: " << counter << std::endl;
```

```
    }
}
```

Dette program vil udskrive:

Hej: 0

Hej: 1

Hej: 2

Hej: 3

Og hvorfor det?

En for-loop består af disse elementer:

for(start; betingelse; opdater)

{

  koden;

}

Når koden skal afvikles udføres ”start” delen én gang, i eksemplet fra før stod der counter = 0, dvs counter sættes til at være 0 inden vi starter loopen.

Derpå checker den ”betingelsen”, i vores tilfælde counter < 4, hvis dette er sandt vil den udføre koden, ellers vil den afslutte loopen og gå videre. Da vi lige har sat counter til at være 0 er counter mindre end 4, derfor vil den udføre koden, i vores tilfælde udskrive ”Hej: ” efterfulgt af counter, dvs 0.

Så udfører den ”updater” delen, i vores tilfælde counter++, dvs den lægger 1 til counter, så counter nu er 1.

Derpå checker den betingelsen igen, dvs counter < 4, det er den stadig, så den udfører koden igen og skriver ”Hej: 1”.

Dette fortætter indtil counter bliver talt op til 4, så hopper den ud af loopen.

Som med if behøver vi ikke {} hvis ”kode” kun er et statement, som i vores eksempel.

Vi kan naturligvis også tælle baglæns:

```
for(i = 3; i >= 0; i—)
```

Som vil tælle 3, 2, 1, 0

Eller tælle de lige tal op til 10:

```
for(i = 0; i < 10; i += 2)
```

Og man kan lave to løkker inden i hinanden.

```
    int i;
    int j;
    for(i = 1; i < 5; i++)
    {
       for(j = 0; j < i; j++)
       {
          std::cout << j  << " ";
       }
       std::cout << std::endl;
```

```
    }
```

Dette vil skrive:

0

0 1

0 1 2

0 1 2 3

Opgaver:

Lav et program der udskriver 3 tabellen.

Lav et program der udskriver hele den lille tabel.

For at få det til at stå pænt under hinanden kan du bruge:

```
std::cout << std::setw(3) << j  << " ";
```

Så kommer j til at fylde mindst 3 tegn, uanset hvor mange cifre der er i j. For at kunne bruge std::setw skal du inkludere iomanip, dvs tilføje denne linie i toppen af programmet:

```
#include <iomanip>
```

Du skal bruge to for-loops, inden i hinanden for at få det til at virke.

Afsnit 7

Indtil nu har vi brugt typen int til vores tal. Men der findes en del andre typer til tal som det kan være gavnligt at bruge. Først er der det problem med int at den kun kan bruges til heltal, dvs. en int kan ikke gemme 1,5. For at kunne gemme decimal tal skal man bruge float, double, eller long double. Til normal brug er double fin. float fylder ikke så meget i hukommelsen men er mere upræcis. long double er mere præcis end double, men den fylder mere og er langsommere end double.

Der er et par ting man skal være opmærksom på når man bruger double. Hvis man skriver:

```
double d = 11/4;
```

Ville man måske forvente at d blev 2,75. Det gør den ikke, d bliver 2. Det er fordi både 11 og 4 er heltal, og så regner den 11/4 ud som heltal, runder ned hvilket giver 2, d sættes til denne værdi.

Den letteste måde at løse problemet er at gange med 1,0 inden du regner:

```
double d = 1.0*11/4;
```

Bemærk at vi skriver 1.0 når vi mener 1,0, det er fordi compileren ”snakker” engelsk, hvor man bruger punktum i stedet for komma.

Det andet problem med double er at de ikke er uendelig nøjagtige, dvs. der er grænser for hvor mange decimaler den kan gemme efter ,

Og endelig er der et problem med udskrivning, hvis du har:

```
double d = 11.0/9.0;
std::cout << d << std::endl;
```

Så vil den nok skrive noget i stil med

1.2222

Hvilket kan være fint, men somme tider vil man gerne have den til at skrive flere eller færre decimaler. Dette kan vi styre med std::setw og std::setprecision, f.ex:

```
#include <iostream>
#include <iomanip>
int main()
{
   double d = 11.0/9.0;
   std::cout << std::setw(4) << std::setprecision(3) << d << std::endl;
}
```

Som vil skrive:

1.22

Også til heltal er der andre typer end int. Der er faktisk en hel række:

char -128 - 127

unsigned char 0 – 255

short -32768 - 32767

unsigned short 0 – 65536

int -2147483648 – 2147483647

unsigned int 0 - 4294967296

long -2147483648 – 2147483647

unsigned long 0 - 4294967296

Det ses at unsigned typerne ikke kan have negative tal, dvs tal mindre end 0.

Det er ikke garanteret at int kan have så store tal, det er ikke garanteret at de kan have tal der er større end dem der gælder for short.

Den væsentligste grund til at man ikke altid bruger den største type er at spare på pladsen i computerens hukommelse.

char er også den type der bliver brugt til at gemme bogstaver i, i f.eks. en std::string, men de kan også bruges til at gemme små tal. Det betyder at man skal passe på når man vil skrive dem ud, for cout tror at man vil udskrive et bogstav og ikke et tal når man udskriver dem. Prøv f.eks. at køre:

```
#include <iostream>
int main()
{
   char a = 72;
   char b = 69;
   char c = 74;
   std::cout << a << " " << b << " " << c << std::endl;
}
```

Den vil nok skrive H E J, fordi den opfatter a, b, c som bogstaver. Hvis vi vil have den til at skrive tallene skal vi bede om at få dem skrevet som f.eks. short:

```
   std::cout << short(a) << " " << short(b) << " " << short(c) << std::endl;
```

Hvis du vil have ét bogstav i din kode skal du skrive dem som 'A', dvs. med ' omkring, lidt på samme måde som med strenge, f.ex:

```
char ch = '@';
```

Den sidste type jeg vil nævne nu er bool. En bool kan kun have to værdier; false eller true. false er 0, true er 1, det kan man se ved at forsøge at skrive dem ud. false er falsk, true er sand.

Man kan f.eks. bruge bool med:

```
bool flag = false;
int i;
std::cin >> i;
if(i > 10)
   flag = true;
```

Her sættes flag til true hvis i er større end 10. Vi kunne også skrive det som:

```
int i;
std::cin >> i;
bool flag = i > 10;
```

Det virker fordi udtrykket i > 10 giver en bool som resultat.

Opgaver:

Prøv at se hvad der sker hvis du laver:

```
unsigned int i = 0;
i--;
```

Og udskriver i

Hvad sker der hvis du laver:

```
unsigned short a = 10000;
a *= 10000;
```

Og udskriver a?

Hvad sker der med:

```
double d = 11.0/2;
int a = d;
```

hvis du udskriver a og d?

Afsnit 8

Vi skal lige se lidt mere på læsning fra tastatur, da det kan være ret besværligt, og kan give mange problemer.

Hvis vi f.eks. ville læse et tal, et bogstav og et navn ville man måske bruge:

```
#include <iostream>
#include <string>
int main()
{
   int tal;
   char bogstav;
   std::string streng;
   std::cin >> tal;
   std::cin >> bogstav;
```

```
    std::getline(std::cin, streng);
    std::cout << tal << std::endl << bogstav << std::endl << streng << std::endl;
}
```

Men det virker ikke, for at forstå hvorfor er vi nød til at forstå lidt mere om hvad der sker.

Først læser vi et tal, så programmet vil læse fra tastaturet så længe brugeren skriver tal, når han så trykker Enter, vil den stoppe med at læse ind i tal. Men karakteren ”ny linie”, som programmet modtog da brugeren trykkede Enter sidder stadig og venter på at blive læst.

Når vi så vil læse bogstav bliver dette tegn puttet ind i bogstav. streng vil dog blive læst korrekt.

Vi kan delvist løse problemet ved at putte denne linie ind efter linjen der læser tal:

```
    std::cin.ignore(1024, '\n');
```

Denne linie vil smide op til 1024 tegn væk, eller indtil den møder ”ny linie”, og så vil resten af programmet (delvist) virke.

Det næste problem er at der måske sidder et ”ny linie” bogstav i bufferen når vi vil læse streng.

For at løse dette problem er vi nød til at læse en hel streng når vi vil læse bogstav, og så tage det først bogstav i streng:

```
    std::string Temp;
    std::getline(std::cin, Temp);
    if(Temp.empty())
        bogstav = '\n';
    else
        bogstav = Temp[0];
```

if/else delen sikrer at det går godt selv om brugeren bare trykker Enter. Temp[0] er det første bogstav i Temp.

Alt i alt vil det skudsikre program se sådan ud:

```
#include <iostream>
#include <string>
int main()
{
    int tal;
    char bogstav;
    std::string streng;
    std::cin >> tal;
    std::cin.ignore(1024, '\n');
    std::string Temp;
    std::getline(std::cin, Temp);
    if(Temp.empty())
        bogstav = '\n';
    else
        bogstav = Temp[0];
    std::getline(std::cin, streng);
    std::cout << tal << std::endl << bogstav << std::endl << streng << std::endl;
```

```
}
```

De to regler som du kan lære af dette er:

1: Put denne linie ind mellem linjer der læses med >> og linjer der læses med getline:

```
std::cin.ignore(1024, '\n');
```

2: Når du skal læse et enkelt bogstav, så læs en std::string og tag det første bogstav.

Opgave:

Lav et program der beder brugeren om to tal, og derpå skal han skrive et bogstav for at fortælle om de to tal skal lægges sammen, trækkes fra hinanden, osv. Dette skal programmet så gøre og skrive resultatet ud.

Afsnit 9

Vi så tidligere at man kunne lave løkker med for(), er findes to andre måder at laver løkker på, dem skal vi kikke lidt på her.

Den første løkke hedder do-while, og ser sådan ud:

```
do
{
   kode
}
while(betingelse);
```

Kode vil blive udført så længe betingelse er sand.

Hvis man vil have brugeren til at skive et tal der er større end 1000, og vil spørge igen og igen indtil han gør det, kunne man bruge:

```
int tal;
do
{
   std::cout << "Skriv et tal der over 1000: ";
   std::cin >> tal;
}
while(tal <= 1000);
```

Den anden løkke hedder bare while og ser sådan ud:

```
while(betingelse)
{
   kode
}
```

Bemærk at der ikke er ; efter while()

Igen hvis vi vil have et tal der større end 1000, men vil fortælle brugeren at han har tastet forkert hvis han skriver noget der ikke er større end 1000 kunne man bruge:

```
int tal;
std::cout << "Skriv et tal der over 1000: ";
std::cin >> tal;
while(tal <= 1000)
```

```
    {
        std::cout << "Tallet er for lille, proev igen: ";
        std::cin >> tal;
    }
```

Forskellen på do-while og while løkker er at do-while altid udfører kode delen mindst én gang, mens while ikke altid udfører løkken, i eksemplet ovenfor vil while løkken ikke køres hvis brugeren inden har skrevet et tal der er større end 1000.

Opgaver:

1: Lav programmet der skrev den lille tabel om så det bruger while.

2: Lav det så det bruger do-while.

3: Et lille spil. Programmet skal vælge et tal som brugeren skal gætte. Det skal fungere ved at brugeren skriver et tal, så skal programmet skrive om det er mindre eller større end det tal der skal gættes og brugeren skal skrive et nyt tal, hvis det er det rigtige tal skal programmet skrive at det var det rigtige tal, og hvor mange gæt brugeren brugte til at gætte tallet.

For at finde et tilfældigt tal skal du bruge rand(), følgende vil give et tilfældigt tal mellem 0 og 99:

int tal = rand()%100;

For at undgå at programmet vælger det samme tilfældige tal skal du bruge srand:

srand(time(0));

Du skal inkludere time.h og stdlib.h. Et eksempel, der vil skrive et tilfældigt tal mellem 0 og 99:

```
#include <iostream>
#include <time.h>
#include <stdlib.h>
int main()
{
    srand(time(0));
    int tal = rand()%100;
    std::cout << tal << std::endl;
}
```

Afsnit 10

Indtil videre har vi puttet al vores koden ind i main. Det er også fint til små programmer, men forestil dig at hele koden til Windows (eller Linux) skulle være inden for main funktionen, det ville være ret upraktisk.

Derfor har man opfundet funktioner, der kan bruges til at splitte koden op i mindre dele.

Lad os se på et eksempel på en funktion:

```
#include <iostream>
int add(int a, int b)
{
    int c = a + b;
    return c;
}
```

```
int main()
{
   int x = add(2, 3);
   std::cout << x << std::endl;
}
```

Her er add en funktion. I parentesen efter navnet på funktionen står parametrene til funktionen, og deres type. De står på ca. samme måde som man laver variabler. Hvis ikke funktionen skal bruge nogen parametre lader man bare parentesen være tom. int foran navnet på funktionen betyder at den returnerer en int, hvis ikke den skal returnere noget skal man skrive void i stedet.

I funktionen laver vi en variabel, c, der er a + b, denne værdi returneres.

I main kalder vi funktionen, dvs udfører koden i funktionen, funktionen laver ikke noget før den bliver kaldt. Da vi har skrevet 2 og 3 i parentesen betyder det at disse bliver til parametre til funktionen. I funktionen vil a således være 2 og b vil være 3. c bliver sat til at være summen af de to, dvs. 5, denne værdi returneres. Tilbage til main vil x blive sat til at være returværdien fra add, dvs. 5.

Det er jo ganske simpelt? Du har allerede brugt flere funktioner, f.eks. std::getline, rand.

Lad os tage et eksempel mere. Vi så før at det var besværligt at læse et enkelt bogstav, men det er måske noget der skal gøres mange gange, så lad os lave en funktion til det:

```
char GetBogstav()
{
   std::string S;
   std::getline(std::cin, S);
   return S[0];
}
```

Så kan man bruge den med.

```
   std::cout << "Vil du? ";
   char ch = GetBogstav();
   if(ch == 'j')
      std::cout << "Ok" << std::endl;
```

Smart ikke?

Der er lige en ting mere vi skal have på plads. For at vi kan få lov til at kalde en funktion skal kompileren vide hvordan funktionen ser ud. I det først eksempel skrev vi funktionen add inden main, og så er der ingen problemer. Men ofte vil man gerne have dem byttet om, så main står først, og hvad gør man så? Man laver en prototype på funktionen:

```
#include <iostream>
int add(int a, int b);
int main()
{
   int x = add(2, 3);
   std::cout << x << std::endl;
```

```
}
int add(int a, int b)
{
   int c = a + b;
   return c;
}
```

Her er den første linie med add en prototype, den ser ud på samme måde som selve funktionen, indtil {, hvor der i stedet er et ; En prototype betyder noget i retning af at, der kommer senere en funktion der ser sådan ud, jeg skal nok fortælle dig hvordan den ser ud senere.

Opgaver:

Lav regnestykke eksemplet fra afsnit 8 om så det bruger funktioner til plus, minus, gange og dividere.

En funktion kan naturligvis godt kalde andre funktioner, og den kan enda kalde sig selv. Lav en funktion der kalder sig selv og beregner 3*2*1 hvor det første tal i rækken skal være en parameter. Du skal bruge en if for at undgå at den kalder sig selv igen og igen for evigt. Hvor store tal kan man gøre det med og få det rigtige resultat? Brug evt. windows lommeregner til at checke, den kan regne med meget større tal end os, der hedder det n!

Afsnit 11

Lad os kikke lidt på dette kode:

```
#include <iostream>
void foo(int a)
{
   a++;
   std::cout << a << std::endl;
}
int main()
{
   int x = 1;
   foo(x);
   std::cout << x << std::endl;
}
```

Hvad vil det skrive? Man kunne tro at det ville skive 2 begge gange, men det gør det ikke. Det skriver 2 første gang og 1 anden gang.

Det skyldes at når vi kalder foo vil den modtage et kopi af x, denne kopi (som er 1) vil den lægge 1 til, resultatet (2) vil den så udskrive. Tilbage i main vil vi udskrive x, den er stadig 1 da foo kun ændrede på kopien af x, ikke x i sig selv.

Det er fint i nogen tilfælde, men i andre tilfælde vil vi gerne have foo til at ændre på x og ikke kun kopien af x, og hvad gør man så? Man bruger en reference.

En reference er en variabel der referer til en anden variabel, man siger også at en reference er et alias, dvs. et andet navn, for en anden variabel. En reference i sig selv kan ikke indeholde noget, den kan kun referere til en anden variabel.

Et simpelt eksempel:

```
#include <iostream>
int main()
{
   int i = 2;
   int& r = i;
   i++;
   std::cout << i << ", " << r << std::endl;
   r++;
   std::cout << i << ", " << r << std::endl;
}
```

Her er i en almindelig int. r er en reference til en int, den sættes til at referere til i.

Når vi så tæller i op, og udskriver både i og r vil de begge skrive at de er 3, simpelthen fordi det er to udgaver af det samme.

Når vi derpå tælle r op, tæller vi i virkeligheden i op, og både i og r vil fortælle at de er 4.

Og nu til bage til vores eksempel, hvis vi laver foo om så den får en reference i stedet:

```
#include <iostream>
void foo(int& a)
{
   a++;
   std::cout << a << std::endl;
}
int main()
{
   int x = 1;
   foo(x);
   std::cout << x << std::endl;
}
```

Så vil programmet skrive 2 begge gange! Det er fordi a, som er parameteren til foo, nu er en reference til x, så nå foo tæller a op, vil den tælle det op som a refererer til, dvs. x i main.

En reference vil altid referere til det samme, man kan ikke sætte den til at referere til noget andet. Og en reference vil altid referere til noget.

De følgende eksempler er ugyldige:

```
int& x = 1;
```

En reference skal referere til en variabel.

```
int& y;
```

En reference skal referere til noget.

```
Int i, j;
int& z = i;
z = j;
```

En reference skal skal altid referere til det samme.

En reference kan være const:

```
double d;
const double& r = d;
```

Man kan godt læse fra en const reference, men man kan ikke skrive til den. Det kan i nogle tilfælde være praktisk, f.eks. i følgende:

```
void foo(const int& a)
{
   std::cout << a << std::endl;
}
int main()
{
   foo(1);
}
```

Hvis ikke a i kaldet til foo var const kunne man ikke bruge 1 som argument, men var nødt til at lave en variabel, det betyder så at foo ikke kan ændre a, men det giver heller ikke rigtigt mening at ændre 1.

Det samme gælder her:

```
void foo(const std::string& a)
{
   std::cout << a << std::endl;
}
int main()
{
   foo("Hello World");
}
```

Når man vil overføre en std::string til en funktion er det oftest en fordel at bruge referencer fordi programmet så slipper for at kopiere strengen. Hvis man så ikke vil have at funktionen roder i ens streng kan man lave den const.

Opgaver:

1: Lav regnestykke eksemplet som vi så på i afsnit 10 om, så funktionerne ikke returnerer resultatet, men overfører det som en parameter vha. reference.

Afsnit 12

Vi så i sidste afsnit at man kunne lave en add funktion:

```
int add(int a, int b)
```

Hvis man så ville bruge funktionen med double, måtte man lave en version der bruger double:

```
double add(double a, double b)
```

Dette ville virke fint, men det er lit trivielt at lave add funktioner for alle mulige typer. Og netop det trivielle vil programører gøre meget for at undgå, så der findes naturligvis noget der er smartere, man kan bruge template funktioner:

```
template <typename T>
```

```
T add(T a, T b)
{
   T c = a + b;
   return c;
}
```

Dette er en template funktion der kan lægge alle typer sammen, man bruger den på samme måde som andre funktioner:

```
   std::cout << add(2, 4) << std::endl;
```

Det der sker er at når compileren skal oversætte finder den ud af hvilen type T er og så laver den en funktion der tager netop denne type.

Man skal være opmærksom på at de to argumenter i dette tilfælde skal være den samme type, og det er også typen på det der returneres.

Hvis vi skriver:

```
   std::cout << add(2.4, 4) << std::endl;
```

Vil compileren brokke sig, for de to argumenter har ikke samme type, 2.4 er en double, 4 er en int. Vi kan løse problemet ved at skrive:

```
   std::cout << add(2.4, 4.0) << std::endl;
```

For så er begge argumenter double, og compileren er glad igen. Eller vi kan fortælle compileren at den skal bruge en bestemt type for T, f.eks:

```
   std::cout << add<double>(2.4, 4) << std::endl;
```

Så vil den lave 4 om til en double og bruge double udgaven af add.

Vi kan naturligvis godt have flere template argumenter:

```
template <typename T1, typename T2>
T1 add(T1 a, T2 b)
{
   return a + b;
}
```

Så behøver de to argumenter ikke have samme type, og funktionen vil returnere en værdi af den type som det første argument havde.

Bemærk at for at kalde en template funktion er det ikke nok med en prototype, compileren skal have hele funktionen for at kunne oversætte.

Opgaver:

Lav regne maskine eksemplet om til at bruge template funktioner. Prøv at ændre typen af det der regnes på fra int til double og unsigned short.

Afsnit 13

Udover at regne er det at arbejde med strenge en af de ting programmet oftest gør, så vi må hellere se lidt mere på hvordan man arbejder med strenge.

Vi har allerede set på at man kan holde en streng i en variabel af typen std::string:

```
std::string S = "Hej Der";
```

Vi kan også sætte strenge sammen:

```
std::string S = "Hej";
```

```
S += " Der";
```

S vil så indeholde ”Hej Der”.

Men hvis vi har:

int x = 12;

Og gerne vil have S til at indeholde "x = 12", hvad gør man så. Man kunne skrive:

```
std::string S = "x = ";
S += x;
```

Men det virker ikke. Det er fordi x er en int og man kan ikke lægge en int til en std::string. Så vi skal selv lave denne int om til en std::string og derpå lægge dem sammen.

Løsningen er at bruge en std::stringstream.

```
#include <iostream>
#include <string>
#include <sstream>
int main()
{
   std::stringstream SS;
   int x = 12;
   SS << "x = " << 12;
   std::string S = SS.str();
   std::cout << S << std::endl;
}
```

En stringstream bruges ca. på samme måde som std::cout, dvs man skriver til den med <<. Men hvor std::cout skriver på skærmen, gemmer en stringstrem det man putter i den i en std::string. Denne kan man så hente med SS.str().

Men da vi ofte vil få brug for at lave en int om til en std::string var det måske en god idé at lave det med en funktion:

```
std::string ToString(int n)
{
   std::stringstream SS;
   SS << n;
   return SS.str();
}
```

For så kan man skrive:

```
int main()
{
   std::string S = "x = ";
   int x = 12;
   S += ToString(x);
   std::cout << S << std::endl;
}
```

Hvilket vist er noget kønnere, eller man kan skrive det lidt kortere:

```
   std::string S = "x = " + ToString(x);
```

Men hvad så med double, short og andre typer? Som du så før er der noget der hedder template funktioner, så det ville være logisk at lave ToString om til at være en template funktion, så den kan håndtere alle typer:

```
template<typename T>
std::string ToString(T n)
{
   std::stringstream SS;
   SS << n;
   return SS.str();
}
```

Og der er jo ret elegant.

Men hvad så med den modsatte vej, hvis man har:

```
   std::string S = "123";
```

Og gerne vil have int x til at være 123? Igen er std::stringstream løsningen:

```
   std::string S = "123";
   std::stringstream SS(S);
   int x;
   SS >> x;
   std::cout << x << std::endl;
```

Men der er en ting som vi lige skal være opmærksomme på. Alle tal kan laves om til en std::string, men det er ikke alle std::string der kan laves om til et tal. Hvis den f.eks. indeholder ”Hej der”, kan man jo ikke lave det om til et tal. Heldigvis fortæller >> om det gik godt:

```
   bool Success = SS >> x;
   if(Success)
      std::cout << x << std::endl;
   else
      std::cout << "Det var ikke et tal: " << S << std::endl;
```

Og da vi jo er i gang med at lave template funktioner var det nok en god ide at lave en:

```
template<typename T>
bool FromString(T& tal, const std::string& str)
{
   std::stringstream SS(str);
   return SS >> tal;
}
```

Denne funktion konverterer str til et tal, og returnerer true hvis str indeholder et tal.

Vi kan så bruge den med:

```
   std::string S = "123";
   int x;
   if(FromString(x, S))
      std::cout << x << std::endl;
   else
```

```
    std::cout << "Det var ikke et tal: " << S << std::endl;
```

Det kan næsten ikke blive lettere.

Vi har set at det er ret let at sætte strenge sammen, men hvordan splitter man dem så ad? For at kunne det har vi brug for to ting: At vide hvor vi vil splitte, og en måde hvorpå vi kan tage en del af en std::string.

Det første kan gøres med en af std::string's mange finde funktioner. En af dem er find, som finder det første tegn af en type i strengen. Et eksempel:

```
std::string S = "Hej Du Der";
std::string::size_type n = S.find(' ');
if(n != std::string::npos)
    std::cout << "Fandt et mellemrum paa position: " << n << std::endl;
```

Da det første mellemrum i "Hej Du Der" er på plads 3, idet vi tæller fra 0, vil n blive sat til at være 3. Hvis ikke der var mellemrum i S ville n blive sat til at være std::string::npos.

Vi kan bruge rfind på samme måde til at finde det sidste tegn i strengen. rfind vil i vores eksempel give 6, da det sidste mellemrum er på plads 6.

På samme måde kan man bruge find_first_of("aeoiyu") til at finde den første vokal i strengen, og find_last_of("0123456789") til at finde det sidste tal i strengen.

Hvis vi vil finde et ord i strengen kan man bruge find("ord") eller rfind("ord").

Når man søger skelnes der mellem store og små bogstaver.

Hvis vi nu vil have det første ord i S over i en anden variabel kan man bruge:

```
std::string S = "Hej Du Der";
std::string::size_type n = S.find(' ');
if(n != std::string::npos)
{
    std::string S2 = S.substr(0, n);
    std::cout << S2 << std::endl;
}
```

substr giver en del af strengen fra en plads, i dette tilfælde 0, og et antal bogstaver frem, i dette tilfælde n, som vi før så var 3. Så S2 bliver sat til at være "Hej".

Hvis vi vil have resten af strengen fra det første mellemrum kan vi skrive:

```
    std::string S3 = S.substr(n + 1);
```

Vi lægger 1 til n for at springe mellemrummet over. Da vi kun bruger én parmeter i kaldet til substr får vi resten.

Opgaver:

1: Lav regnemaskinen om så regnestykket læses som én streng, f.eks. "123 + 321". Du skal så splitte denne streng op i tre, det første tal, +-* eller / og det andet tal. Du kan bruge den

FromString funktion som vi lavede før, til at lave disse tal om til int.

2: Vi så tidligere at det ikke var nogen god ide at bruge >> til at læse tal med, hvis brugeren skrev noget der ikke var et tal. Løsningen på det problem er at bruge std::getline til at læse en std::string, og så bruge samme metode som i FromString til at lave dette om til et tal. Lav derfor en template funktion der gør dette. Den kommer til at ligne FromString men tager kun et argument, det tal som der bliver læst.

Afsnit 14

Fra tid til anden har man brug for at give ting numre, det kunne f.eks. være ugedage, hvor mandag kunne være nummer 0, tirsdag nummer 1, osv. Man kan selvfølgelig gå og huske på hvad der er hvad, men der findes en mere elegant løsning:

```
enum UgeDag
{
   mandag,
   tirsdag,
   onsdag,
   torsdag,
   fredag,
   loerdag,
   soendag
};
```

Dette hedder en enum, som er en forkortelse af enumeration, der betyder noget i retning af "optælling". I dette eksempel vil mandag være 0, tirsdag 1, osv. Vi kan bruge dette i vores kode:

```
int Dag = mandag;
if(Dag == onsdag)
```

UgeDag fra vores eksempel er faktisk en ny type, så man kan også skrive:

```
UgeDag Dag = fredag;
```

Hvilket kan være meget praktisk.

Hvis man ikke gør noget specielt for de enkelte værdier i en enum bare fortløbende numre. I nogle tilfælde kan det være praktisk at give dem specielle værdier. Hvis vi f.eks. vil gøre en ting for alle ugedage kunne man skrive:

```
int i;
for(i = mandag; i <= soendag; i++)
   Whatever(i);
```

Men det forudsætter at vi husker på at mandag er den første dag i ugen, og søndag den sidste. Man kan løse det ved at tilføje to værdier til enum'en:

```
enum UgeDag
{
   Mandag,
   Tirsdag,
   Onsdag,
   Torsdag,
   Fredag,
   Loerdag,
   Soendag,
   FoersteUgeDag = Mandag,
   SidsteUgeDag = Soendag
};
```

For så kan man skrive:

```
int i;
for(i = FoersteUgeDag; i <= SidsteUgeDag; i++)
   Whatever(i);
```

Nu sidder du måske og tænker, hvorfor bruge han ikke UgeDag som type for i? Men det virker ikke, fordi ++ ikke virker med enum's.

Man kan også give de enkelte værdier i enum'en en talværdi:

```
enum DageIMaaned
{
  DageIJanuar = 31,
  DageIFebruar = 28,
  DageIFebruarSkudAar = 29,
```

Enum kan antage alle heltals værdier.

Hvis man skulle skrive en funktion der udskriver navnet på en ugedag, kunne man skrive koden som:

```
void SkriveUgedag(UgeDag dag)
{
   if(dag == mandag)
      std::cout << "mandag";
   else if(dag == tirsdag)
      std::cout << "tirsdag";
```

Og så videre. Men der er en mere elegant måde at gøre det på:

```
void SkriveUgedag(UgeDag dag)
{
   switch(dag)
   {
   case mandag:
      std::cout << "mandag";
```

```
      break;
   case tirsdag:
      std::cout << "tirsdag";
      break;
```

Det kaldes en switch, det fungerer på den måde at man hopper fra switch'en til den case som passer med det man switcher på. Og udfører koden indtil den møder break, så hopper den ud.

Man kan switch'e på alle heltal, også bogstaver:

```
   char ch = GetBogstav();
   switch(ch)
   {
   case 'j';
      std::cout << "Ok" << std::endl;
      break;
   case 'n';
      std::cout << "Naa ikke" << std::endl;
      break;
   default:
      std::cout << "Det forstod jeg ikke" << std::endl;
      break;
   }
```

I dette tilfælde vil den hoppe til default hvis ikke ch er j eller n. Man kan lave alt det kode man vil efter case, det behøver ikke kun være en linie. I nogle tilfælde springer man break over:

```
   int n;
   GetTal(n);
   switch(n)
   {
   case 2;
      std::cout << "Hej Verden" << std::endl;
   case 1;
      std::cout << "Hej Verden" << std::endl;
      break;
   case 0:
      break;
   default:
      std::cout << "Det forstod jeg ikke" << std::endl;
      break;
   }
```

Her vil den skrive "Hej Verden" to gang hvis n er 2, 1 gang hvis den er 1, 0 gange hvis den er 0, og "Det forstod jeg ikke" i alle andre tilfælde. Men som regel er det en forglemmelse hvis ikke der står break, så husk det nu.

Opgvave: Lave en enum for de fire regnearter. Lav regneprogrammet om så det laver bogstavet for regnearten om så det bruger denne enum. Du kan lave det så plus = '+', så behøver du ikke oversætte. Brug så en switch til at vælge hvilket regnestykke der skal laves.

Afsnit 15

Vi har set på hvordan man kan oprette variable til at indeholde ét tal eller én streng. Men ofte har man brug for flere af disse for at beskrive en "ting". Hvis vi f.eks. arbejder med grafik på en flade, ville vi bruge en x og en y til at beskrive et punkt. Ville vi lave en telefonbog skulle den mindst indeholde navn og telefon nummer. Den simple måde at samle sådan information i C++ er vha. en struct. Hvis vi tager eksemplet med punktet fra før kunne man gøre det på denne måde:

```
struct Punkt
{
   int x;
   int y;
};
```

Her har vi lavet en ny type kaldet Punkt, der indeholder to variable, x og y. Vi kan så oprette en variabel af denne type:

```
Punkt p;
```

p er her variablen, man siger også at p er et objekt.

Vi kan nu få fat i p's x og y med .

```
Punkt p;
p.x = 123;
p.y = 321;
std::cout << p.x << ", " << p.y << std::endl;
```

Man siger at x og y er medlemmer af Punkt, eller medlems variable.

Variabler af typen Punkt opfører sig på samme måde som andre variabler:

```
#include <iostream>
struct Punkt
{
   int x;
   int y;
};
void Skriv(const Punkt& p)
{
   std::cout << p.x << ", " << p.y << std::endl;
}
int main()
```

```
{
   Punkt p1 = {2, 5};
   Punkt p2;
   p2.x = 2;
   p2.y = 5;
   Skriv(p1);
   p2 = p1;
   Skriv(p2);
}
```

Bemærk måden vi tilskriver p1 dens værdi, man skriver værdierne i samme rækkefølge som de står i struct'en.

Vi kan også sætte den ene til at være lig den anden, men vi kan ikke umiddelbart sammenligne dem med ==

Skulle vi lave en telefonbog kunne vi lade hver opslag se sådan ud:

```
struct Entry
{
  std::string Navn;
  unsigned int Nummer;
};
```

Selv om man nok i praksis vil bruge en std::string til nummer, for at det kan indeholde mellemrum og andre special tegn.

Der er også en standard struct det er rart at kende, den kaldes tm, og bruges til at holde styr på et tidspunkt. Den kan bruges med:

```
#include <iostream>
#include <time.h>
int main()
{
   tm Time;
   memset(&Time, 0, sizeof(Time));
   Time.tm_mday = 12;
   Time.tm_mon =  6;
   Time.tm_year = 2006 - 1900;
   Time.tm_hour = 11;
   Time.tm_min = 10;
   Time.tm_sec = 0;
   std::string Tid = asctime(&Time);
   std::cout << Tid << std::endl;
}
```

Time er objektet af typen tm. memset bliver brugt til at nulstille den, det er nødvendigt, ellers vil programmet (måske) bryde ned når du kører det.

De næste linier bliver brugt til at sætte de enkelte fælter i Time. Bemærk at den regner år 1900 som år 0, så du skal trække 1900 fra årstallet for at få det rigtige resultat. tm_mday er dag i måned, første dag er 1. tm_mon er måned, første måned, januar, er, sært nok, 0.

Kaldet til asctime laver tiden i tm format om til en streng, denne udskrives, resultatet kunne se sådan ud:

Sun Jul 12 11:10:00 2006

Man kan beregne forskellen på to tider i tm format, vha. difftime funktionen, men først skal man lave tiden om til time_t format:

```
tm Time1;
tm Time2;
time_t t1 = mktime(&Time1);
time_t t2 = mktime(&Time2);
double diff = difftime(t1, t2);
std::cout << diff << std::endl;
```

Forskellen er i sekunder.

Man kan hente den nuværende tid vha. time, man får tiden i time_t format:

```
time_t t1;
time(&t1);
```

Man kan lave denne time_t om til en tm med localtime:

```
tm Time1 = *localtime(&t1);
std::cout << asctime(&Time1) << std::endl;
```

Dvs localtime laver det modsatte af mktime.

Hvis du undrer dig over hvad de & der bruges i diverse kald og * foran localtime betyder, så må du indtil videre nøjes med at vide at de skyldes at der arbejdes med pointere. Jeg skal nok forklare dem senere.

Opgaver:

1: Lav en funktion der beder brugeren om at indtaste sin fødselsdag, og få så programmet til at skrive hvor mange sekunder og timer han er gammel. Det vil nok være praktisk at lave en funktion der beder brugeren om fødselsdag, og så lave resten i en anden funktion.

Afsnit 16

En lidt anden måde at lave telefonbogs opslaget fra sidste afsnit var at bruge en class:

```
class Entry
{
```

```
public:
  std::string Navn;
  unsigned int Nummer;
};
```

Når den skrives på denne måde vil den virke på samme måde som den med struct. I et senere afsnit skal jeg nok fortælle hvad public: gør.

Vi kunne så lave en funktion der læser et Entry:

```
void ReadEntry(Entry& entry)
{
   std::getline(std::cin, entry.Navn);
   std::cin >> entry.Nummer;
}
```

Og man kunne lave en tilsvarende funktion til at udskrive dem. Men, der findes en smartere måde, man kan lave en funktion i class'en, så den kommer til at se sådan ud:

```
class Entry
{
public:
  void ReadEntry();
  std::string Navn;
  unsigned int Nummer;
};
```

Her er ReadEntry blevet til en medlems funktion, det er kun en en prototype, selve funktionen kan vi lave uden for class'en, den kunne se sådan ud:

```
void Entry::ReadEntry()
{
   std::getline(std::cin, Navn);
   std::cin >> Nummer;
}
```

Det bemærkes at man skriver Entry:: foran funktions navnet, det er for at markere at det er Entry's ReadEntry, da andre class'er kan have funktioner der hedder det samme. Det bemærkes også at vi bare skriver Navn, det er fordi member funktioner godt ved at det er dens egen Navn der menes, så vi behøver ikke at fortælle den det. Det samme gælder for Nummer.

Når vi så vil kalde denne ReadEntry funktion gør man sådan her:

```
Entry entry;
entry.ReadEntry();
```

I kaldet til entry.ReadEntry vil den så læse entry.Navn og entry.Nummer.

Hvis man vil sammenligne to Entry's er man nød til at lave en funktion til det. Man kunne gøre det med en medlems funktion, så class'en kom til at se sådan ud:

```
class Entry
{
public:
  void ReadEntry();
  bool Compare(const Entry& other);
  std::string Navn;
  unsigned int Nummer;
};
```

Compare funktionen kunne se sådan ud:

```
bool Entry::Compare(const Entry& other)
{
   if(Navn == other.Navn && Nummer == other.Nummer)
      return false;
   return true;
}
```

Og man kunne bruge den sådan her:

```
   Entry e1, e2;
   if(e1.Compare(e2))
      std::cout << "..." << std::endl;
```

I Compare ser vi at den ene udgave af Entry kan tilgåes ved blot at skrive Navn, det er e1 fra main, den anden udgave af Entry, e2 tilgår vi ved at skrive other.Navn.

Senere skal vi se hvordan man kan lave det så man kan sammenlige med ==

Opgave:

1: Lave en Udskriv funktion til Entry ovenfor.

2: Endnu en gang regnemaskinen. Lav en class som udgør et regnestykke. class'en skal have tre funktioner, én til at bede brugeren om det der skal beregnes, én til at beregne og én til at udskrive.

Afsnit 17

Der er to funktioner i class'er som er specielle.

Den første er den funktion (eller rette de funktioner) der hedder det samme som class'en. Det specielle ved denne funktion er at den bliver kaldt når man laver en variabel eller objekt af class'en.

Et meget simpelt eksempel:

```
#include <iostream>
class X
{
public:
   X()
   {
      std::cout << "Hej der" << std::endl;
   }
};
int main()
{
   X x;
}
```

Dette program vil, når det køres skrive "Hej der", fordi vi opretter en X, og når vi gør det køres X::X.

Bemærk at vi i dette tilfælde skrev selve koden til funktionen inden i class'en, det er meget almindeligt for små funktioner som denne.

X::X er en constructor, som er et engelsk ord, der kan oversættes til konstruktør, altså en der opretter objektet.

Bemærk også at constructorer ikke returnerer noget, man kan og skal altså ikke skrive int eller void foran.

Ofte bruger man constructoren til at sætte default værdier for objektet, vi kunne f.eks. lave:

```
#include <iostream>
class X
{
public:
   X(int y)
   {
      x = y;
   }
   int x;
};
int main()
{
   X x(123);
}
```

Man kan også skrive X::X sådan:

```
   X(int y) : x(y)
   {
   }
```

Det gør det samme som den X::X som vi så før. Tricket kan kun bruges med constructorer.

Den anden funktion er er speciel er funktionen (og der er kun én) der hedder det samme som class'en men med ~ foran. Du laver en ~ ved at trykke AltGr+<knap til højre for Å> og mellemrum.

Et simpelt eksempel:

```
#include <iostream>
class X
{
public:
   X()
   {
      std::cout << "Goddag" << std::endl;
   }
   ~X()
   {
      std::cout << "Farvel" << std::endl;
   }
};
int main()
{
   X x;
}
```

Dette program vil skrive:

Goddag

Farvel

Det skriver Goddag når programmet når til linjen X x; Og det skriver Farvel når det kommer til den sidste },

destructor bruges normalt til at rydde op efter sig selv, det skal vi kikke på senere.

Opgaver:

1: Lav regnemaskinen fra sidste afsnit om så det er constructoren der beder brugeren om de tal der skal beregnes, og destructoren skal skrive resultatet.

2: Lav regnemaskinen om så de to tal overføres i constructoren.

3: Vha. difftime osv. som vi så på i forrige afsnit skal du lave en class der når den bliver oprettet husker hvornår den blev oprettet. Lav så to udgave af denne class, med en std::cin.get()

imellem, derved kan du beregne hvor mange sekunder brugeren var om at trykke enter. Lav denne beregning i en member funktion.

Afsnit 18

Vi lavede tidligere denne class:

```
class Entry
{
public:
  void ReadEntry();
  bool Compare(const Entry& other);
  std::string Navn;
  unsigned int Nummer;
};
```

Men at bruge compare til at sammenligne med var lidt kluntet, det kunne være smartere at man kunne skrive:

```
   Entry e1, e2;
   if(e1 == e2)
      std::cout << "..." << std::endl;
```

For at kunne gøre det er vi nødt til at lave en operator. En operator virker næsten på samme måde som en funktion, den ser lidt anderledes ud og man kalder den på en anden måde, men ellers er det det samme.

En == operator kunne se sådan ud:

```
bool operator == (const Entry& lhs, const Entry& rhs)
{
   if(lhs.Navn == rhs.Navn && lhs.Nummer == rhs.Nummer)
      return false;
   return true;
}
```

Bemærk at denne operator ikke er medlem af class'en, det er en global funktion. lhs bliver ofte brugt for navn på den ene parameter, lhs betyder ”Left Hand Side”, dvs. den del der står på venstre side, på samme måde betyder rhs ”Right Hand Side” altså det der står til højre.

Vi kunne også lave operatoren som en del af class'en, så kunne det komme til at se sådan ud:

```
class Entry
{
public:
  std::string Navn;
  unsigned int Nummer;
  bool operator == (const Entry& rhs)
  {
```

```
    if(Navn == rhs.Navn && Nummer == rhs.Nummer)
       return false;
    return true;
  }
};
```

Bemærk at vi her kun får et argument, det andet argument er automatisk lhs.

Om man foretrækker den ene eller den anden form er mest et spørgsmål om smag og behag.

Man kan lave operatorer for alle de almindelige operationer, f.eks, + - * / = += ++ osv, men man kan ikke opfinde sine egne, f.eks. kan man ikke lave en <> eller en +- operator.

Opgaver:

1: lav en class der indeholder et tal, og funktioner til at læse og skrive dette tal. Lav så +,-,* og / operator for to objekter af denne class. Derved kan du lave endnu en version af regnemaskinen.

Afsnit 19

Vi har set hvordan man kan bruge en variable til at gemme et tal eller en streng. Med struct og class kunne man også lave en variabel der indeholder flere af disse. Man kunne, som vist ovenfor, lave en class der indeholder et opslag i en telefonbog. Men hvis man vil lave en telefonbog var det praktisk at have en liste eller en anden form for samling over de enkelte opslag i telefonbogen.

Da det er et almindeligt problem, og da der findes mange måder at løse det på med hver sine fordele og ulemper, findes der naturligvis en række metoder.

Vi skal her se på en enkelt af disse, det er en std::vector, som er en af mange standard containere, som har en del fællestræk.

Hvis vi har en class kaldet Entry som ovenfor, kan vi lave en mængde af disse med:

```
std::vector<Entry > TelefonBog;
```

Når bogen er oprettet på denne måde er den tom, men vi kan fylde nogen i:

```
Entry entry;
TelefonBog.push_back(entry);
```

Dette vil putte entry i listen som det sidste. Man kan putte lige så mange Entry's i bogen som din computer har plads til.

Du skal huske at inkludere vector filen for at kunne bruge vector:

```
#include <vector>
```

Hvis Entry har en funktion til at udskrive kan vi udskrive hele listen med:

```
unsigned int n;
for(n = 0; n < TelefonBog.size(); n++)
   TelefonBog[n].Udskriv();
```

Det ses at TelefonBog.size() fortæller hvor mange opslag der er i bogen.

Man kan få det n'te opslag med TelefonBog[n]

En anden måde at udskrive på:

```
std::vector<Entry >::iterator it;
for(it = TelefonBog.begin(); it != TelefonBog.end(); it++)
   it->Udskriv();
```

En iterator er en dims til at gennemløbe en container, i dette tilfælde en vector. Den opfører sig lidt som en reference til et element i vectoren, men kan flyttes frem. TelefonBog.begin() er det første element i telefonbogen. TelefonBog.end() er elementet lige efter det sidste element i bogen, der står altså ikke noget der.

Hvis vi vil slette et opslag kan vi bruge:

```
TelefonBog.erase(TelefonBog.begin() + 10);
```

Hvor 10 er indexet for det opslag vi vil slette.

Hvis telefonbogen bliver stor er det rart at få den sorteret så det er let at søge i den, det kan man gøre med:

```
std::sort(TelefonBog.begin(), TelefonBog.end());
```

Det kræver at den ved hvordan man ordner en to opslag, det fortæller man den ved at lave en mindre end operator:

```
bool operator < (const Entry& lhs, const Entry& rhs)
```

Som burde være let at lave ved at sammenligne de to navne og telefon numre.

For at bruge std::sort skal du inkludere algorithm:

```
#include <algorithm>
```

Opgave:

Lave et program som beder brugeren om navn og telefonnummer, gem dette i et Entry objekt i en std::vector som ovenfor. Hver gang der er puttet et nyt navn i listen skal listen sorteres og udskrives. Når listen er udskrevet skal brugeren spørges om han vil putte flere i listen, vil han det startes forfra.

Funktionerne til at læse og skrive en opslag skal være members af Entry.

Afsnit 20

At have en telefonbog hvor man hver gang skal taste det hele ind, er ikke ret sjovt. Så det kunne være rart at kunne gemme den i en fil.

Heldigvis er det ret let at læse fra og skrive til filer, det

foregår på samme måde som når man læser fra tastaturet, og skriver til skærmen.

Hvis man vil åbne fil og læse et tal fra den kan man:

```
#include <fstream>
#include <iostream>
int main()
{
   std::ifstream Fil("test.txt");
   if(!Fil)
   {
      std::cout << "Kunne ikke åbne test.txt" << std::endl;
      return 1;
   }
   int i;
   Fil >> i;
   Fil.close();
   std::cout << i << std::endl;
}
```

Fil er en stream af typen ifstream, i betyder her input, dvs. filen kan bruges til at læse fra.

Linien med if(!Fil) finder ud af om filen kunne åbnes. Hvis filen ikke fandes kan den ikke åbnes.

Derpå læses en int (i) på samme måde som med std::cin.

Fil.close() lukker filen igen, det er ikke altid nødvendigt, Fil's destructor vil lukke den hvis vi ikke har gjort det. Da en fil kun kan være åben én gang kan det være nødvendig at lukke den når vi er færdig.

Hvis vi vil skrive til en fil:

```
#include <fstream>
#include <iostream>
int main()
{
   std::ofstream Fil("test.txt");
   if(!Fil)
   {
      std::cout << "Kunne ikke åbne test.txt" << std::endl;
      return 1;
   }
   Fil << 123;
}
```

Bemærk at vi bruger ofstrem, da vi vil skrive, o for output. Hvis ikke filen findes vil den blive oprettet, hvis den findes vil det der stod i den blive slettet, så pas på med hvad du laver.

Opgave: Udvid telefonbogs programmet fra før så det starter med at læse indholdet af telefonbog.txt, og putte det i den TelefonBog. Når programmet afsluttes skrives hele telefonbogen tilbage til telefonbog.txt. Lave to nye funktioner i Entry til at læse fra og skrive til en fil.

Afsnit 21

I det program som du lavede i sidste afsnit har du to funktioner til at læse et Entry, en til at læse fra fil og en til at læse fra tastaturet, på samme måde har du to funktioner til at skrive, til hhv. fil og skærm.

Og som du nok har bemærket er programmører ikke meget for at have flere ting der gør stort set det samme, De finder på smarte måder at omgå det.

Vi vil her lave to operatorer, en til at læse en entry:

```
Entry entry;

std::cin >> entry;
```

Og en til at skrive:

```
Entry entry;

std::cout << entry << std::endl;
```

Det smarte ved disse operatorer er at de også kan bruges med filer. Det virker fordi både std::cin og en std::ifstream er istreams, og fordi både std::cout og std::ofstream er ostreams, vi skal senere se på hvodan det hænger sammen.

Først operatoren til at skrive et entry:

```
std::ostream& operator << (std::ostream& os, const Entry& entry)
{
   os << entry.Navn << std::endl << entry.Nummer;
   return os;
}
```

På samme måde kan vi lave en til at læse:

```
bool operator >> (std::istream& is, Entry& entry)
{
   return std::getline(is, entry.Navn) && is >> entry.Nummer;
}
```

Dette er jo ikke så svært.

Opgave: Opgaven er naturlig nok at lave programmet fra sidste afsnit om til at bruge de to operatorer i stedet for de fire funktioner.

Afsnit 22

I afsnit 15 lovede jeg at fortælle hvad * og & skulle gøre godt for, så det må jeg hellere gøre.

For at computeren kan holde styr på dens hukommelse giver den hver celle i hukommelsen en adresse, en adresse kan siges at være et tal der er et index til en celle. Når vi laver en variabel gemmer computeren den i sin hukommelse på en bestemt adresse. Vi kan få fat i den adresse:

```
int i;
int* p = &i;
```

&i betyder adressen på i, p er en pointer, der indeholder adressen på i. Det kan vi bruge til at ændre i:

```
*p = 123;
```

Dette vil ændre i, da p peger på i. *p betyder indholdet af p, dvs. det p peger på.

En pointer ligner på flere måder en reference, idet de begge er associeret til en anden variabel. Men en pointer er mere fleksibel end en reference, idet:

1: Den kan pege på ingenting:

```
int* p = 0;
```

Hvis man vil finde ud af om en pointer peger på ingenting:

```
if(p == 0)
  std::cout << "p peger ikke på noget" << std::endl;
```

2: Den kan sættes til at pege på noget andet:

```
int i, j;
int *p = &i;
if(i == 7)
   p = &j;
```

Hvis man vil skrive hvad p peger på skal man skrive * foran:

```
std::cout << *p << std::endl;
```

Hvis man bare skriver:

```
std::cout << p << std::endl;
```

Får man adressen, og den er meget sjældent brugbar.

Hvis en pointer peger på et objekt af en class skal man bruge -> i stedet for .

```
Entry entry;
Entry* p = &Entry;
p->Navn = "Ole Hansen";
```

Man kan også sætte pointere til at pege på ”frie” objekter oprettet til formålet:

```
int *p = new int;
```

```
*p = 123;
```

Her opretter vi en int med new i ”free store” også kaldet heap'en.

new er en operator der operetter et objekt af en bestemt type og returnerer en pointer til objektet.

En af ulemperne ved denne metode er at man selv skal nedlægge objekter der er oprette med new, det gøres med delete:

```
int *p = new int;
*p = 123;
delete p;
```

For hver new skal du også have en delete. Hvis du glemmer delete kan vil dit program, hvis det får lov til at køre længe nok, kunne ”spise” alt hukommelsen på din computer, så pas på.

Man kalder brugen af new og delete for ”dynamisk memory alokering”, fordelen ved at bruge dynamisk memory er at man selv kan bestemme hvornår objekter oprettes og nedlægges.

Et eksempel:

```
Entry Get()
{
  Entry e;
  return e;
}
```

I Get opretter vi et Entry og returnerer det. Men i virkeligheden er det ikke e der bliver returneret, men et kopi af e. Hvis e fylder meget i hukommelsen kan det tage lang tid at kopiere et Entry, så man kunne bruge:

```
Entry* Get()
{
  Entry* e = new Entry;
  return e;
}
```

Her er det en pointer til et Entry der returneres, så der sker ingen kopiering. Ulempen er igen at den der kalder Get skal huske at delete det Get returnerer, f.eks:

```
Entry* p = Get()
delete p;
```

Man ser også at man bruger pointere i kald til funktioner:

```
void Pop(int* p)
{
  *p = 123;
}
```

Her fungerer det på samme måde som med referencer. Man skal så kalde funktionen med:

```
int k;
Pop(&k);
```

Det er den metode der blev brugt med memset og asctime tidligere. Men hvis du kan så brug referencer.

Opgave:

Lav en af udgaverne af regnemaskinen om så den kun bruger pointere og ikke "rigtige" objekter, dvs. alle variable skal laves med new. Husk at rydde op efter dig.

Afsnit 23

I nogle tilfælde har man brug for et fast antal af en type. Det kunne være ti int. Så kan man bruge en std::vector, eller man kan lave et array:

```
int Liste[10];
```

Dette vil oprette en liste af 10 int, disse kan man tilgå med:

```
int i;
for(i = 0; i < 10; i++)
   Liste[i] = i;
```

Som med en std::vector hedden den første 0 og den sidste hedder N-1, hvis der er N i arrayet, N er 10 i vores eksempel, så den sidste hedder Liste[9].

I nogle tilfælde ved man ikke på forhånd hvor mange man skal bruge, så kan man oprette den med new []

```
int x = 12;
int* Set = new int [x];
```

Her er der plads til 12 i Set, man bruger den på samme måde som Liste ovenfor, dvs:

```
 Set[0] = 123;
```

Som sædvanligt når vi har brugt new til at oprette noget, skal vi bruge delete til at rydde op. Da vi brugte new[] skal vi også bruge delete[];

```
  delete [] Set;
```

Man siger at et array der er oprettet med new [] er "Dynamisk Array"

En anden anvendelse af pointere er til at gennemløbe et array, lidt på samme måde som en iterator:

```
    int* Set = new int [10];
    int* p;
    for(p = Set; p != &Set[10]; p++)
      *p = 123;
```

I for delen sætter vi p til at pege på Set, dvs. det første element i Set. &Set[10] er adressen på elementet lige efter det

sidste i Set. *p betyder indholdet af p, vi kunne også skrive p[0] = 123; Dette virker med rigtige arrays så vel som med dynamiske arrays. Det ses også at pointere og arrays virker på lidt samme måde.

At pointere virker lidt som en iterator gør at man kan bruge standard funktioner med arrays på samme måde som med std::vector. F.eks. kan man sortere et array:

```
int P[10];
std::sort(P, &P[10]);
```

En af ulemperne ved at bruge dynamiske arrays og pointere er at man ikke kan spørge dem om hvor store de er, dvs hvor mange elementer der er i deres array, for de ved det ikke.

Opgaver:

Lav et program der spørger brugeren om hvor mange tal han vil indtaste, lav så et dynamisk array til disse tal og bed brugeren om at indtaste dem. Sorter derpå array'et og udskriv summen af alle tallene, det største og det mindste tal.

Afsnit 24

Hvis man har lavet en class som kan et eller andet, men i nogle tilfælde ønsker man at få den til at kunne noget mere, kan man naturligvis kopiere den og tilføje det man ønsker.

En anden, og lidt mere elegant måde er at bruge arv. Hvis vi tager Entry som eksempel, men ønsker at tilføje email adresse til den kunne man lave:

```
class Entry
{
public:
  std::string Navn;
  unsigned int Nummer;
  void Udskriv(std::ostream& os)
  {
     os << Navn << std::endl << Nummer << std::endl;
  }
};
class EntryEx : public Entry
{
public:
   std::string EmailAdresse;
};
```

Så vil EntryEx kunne det samme som Entry, men den har også EmailAdresse. Vi siger at EntryEx arver fra Entry, og at Entry er en base class for EntryEx.

Men hvis vil vil udskrive et EntryEx objekt vha. Udskriv

funktionen, vil den kun udskrive Navn og Nummer, for Udskriv er en funktion i Entry, og den kender ikke EmailAdresse. Løsningen er at lave en Udskriv funktion i EntryEx:

```
class EntryEx : public Entry
{
public:
   std::string EmailAdresse;
   void Udskriv(std::ostream& os)
   {
      Entry::Udskriv(os);
      os << EmailAdresse << std::endl;
   }
};
```

Da Udskriv funktionen findes i både Entry og EntryEx vil det være EntryEx udgaven af Udskriv der bliver kaldt. Man siger at Udskriv i den class der arver fra en base class overloader funktionen.

I EntryEx::Udskriv kalder vi først Entry::Udskriv, dvs. Udskriv i base class'en, denne vil udskrive Navn og Nummer, vi skriver derpå EmailAdresse.

Opgave:

Gør koden herover færdig så den kan bruges til at gemme, hente og vise en hel telefonbog med email adresser.

Afsnit 25

I dette afsnit skal vi se lidt på exceptions. Exceptions bruges normalt når noget i programmet går galt, dvs. at noget ikke kan lade sig gøre eller ikke gik som det skulle.

Brug af exceptions er delt i tre dele, en try, en trow og en catch. Dvs. man forsøger noget (try), går det ikke godt er der nogen der kaster (throw) en exception, og en der fanger (catch).

Et simpelt eksempel kunne være en divider funktion der kaster en exception hvis man forsøger at dividere med 0:

```
#include <iostream>
int Divider(int a, int b)
{
   if(b == 0)
   {
      throw "Kan ikke dividere med 0" ;
   }
   return a/b;
}
int main()
```

```
{
   try
   {
      std::cout << Divider(123, 0);
   }
   catch (const char* e)
   {
      std::cout << "Ups, exception: " << e << std::endl;
   }
}
```

I main laver vi en try-blok, i denne kalder vi funktionen Divider.

I Divider checker vi om b er 0, hvis den er, kaster vi en exception i form at en streng, eller rettere en const char *

I main vil vil så fange denne exception, hvis den bliver kastet og udskriver teksten og exceptionen.

Man kan kaste exceptions af alle typer, også int. Men den der fanger skal vide hvad det er der forsøges fanget, eller man kan bruge ... i sin catch, for at fange alt:

```
  catch ()
   {
      std::cout << "Ups, exception" << std::endl;
   }
```

Ulempen ved denne metode er at man ikke kan vide hvad det er der blev kastet.

Der er en del standard funktioner der kaster en exception hvis noget går galt. Hvis man f.eks. bruger en std::vector, men ikke helt har styr på hvor mange elementer der er i den, kan man bruge at() til at hente med, den vil kaste en std::out_of_range exception hvis man forsøger at tilgå et invalid index. F.eks:

```
#include <iostream>
#include <exception>
#include <vector>
#include <stdexcept>
int main()
{
   std::vector<int > MyVector;
   MyVector.push_back(123);
   MyVector.push_back(321);
   MyVector.push_back(111);
   try
   {
      int i = 0;
      while(1)
```

```
            std::cout << MyVector.at(i++) << std::endl;

      }
      catch (std::out_of_range e)
      {
         std::cout << "Ups, exception: " << e.what() << std::endl;
      }
}
```

Dette program vil udskrive alle elementer i MyVector, og exception teksten når den kommer til enden. Med Borland compileren vil programmet skrive:

```
Ups, exception: index out of range in function: vector:: at(size_t)
 index: 3 is greater than max_index: 3
```

Hvis man bruger at funktionen uden at have en try/catch vil programmet sandsynligvis udskrive en fejlbesked og lukke.

Opgave:

Lav telefonbogs programmet om så det bruger exceptions, hvis der indtastes et telefon nummer der ikke er gyldigt. Lav den GetTal funktion du lavede i afsnit 13 om så den kaster en exception hvis det tal brugeren skrev ikke er et gyldigt tal.

Afsnit 26

Du har set at når vi laver class'er skriver jeg public: i toppen og jeg bruger også public ved arv, her skal vi kikke lidt på hvad det betyder.

Hvis du laver en class der skal repræsentere en firkant kunne den se sådan ud:

```
class Firkant
{
public:
   Firkant(int h, int w) :
      Height(h),
      Width(w)
   {}
   int Height;
   int Width;
};
```

Men hvis du så vil lave en funktion der kan fortælle dig arealet af firkanten, kan du enten lave en areal variabel og udregne denne i constructor'en eller du kan lave en funktion der hedder Areal som beregner den hver gang.

Men hvis du ikke helt ved hvilken metode der er bedst, og gerne

vil kunne ændre det senere uden at skrive en mængde kode om, hvad gør du så?

Du kunne starte med at lave en variabel og en funktion:

```
class Firkant
{
public:
   Firkant(int h, int w) :
      Height(h),
      Width(w),
      Areal(h*,w
   {}
   int GetAreal() { return Areal; }
   int Height;
   int Width;
   int Areal;
};
```

Men nu risikerer du at folk der bruger din class bruger Areal direkte, og så skal du skrive koden om, hvis du senere bestemmer dig for at fjerne den. Løsningen er at gøre den private:

```
class Firkant
{
public:
   Firkant(int h, int w) :
      Height(h),
      Width(w),
      Areal(h*,w
   {}
   int GetAreal() { return Areal; }
   int Height;
   int Width;
private:
   int Areal;
};
```

Ved at gøre Areal privat, kan andre ikke bruge Areal, kun class'en selv.

Det betyder at man ikke kan:

```
   Firkant firkant(123, 321);
   std::cout << firkant.Areal << std::endl;
```

Men man bliver nødt til at bruge GetAreal til at hente Areal med:

```
   Firkant firkant(123, 321);
   std::cout << firkant.GetAreal() << std::endl;
```

Nå vi brugte public ved arv:

```
class EntryEx : public Entry
{
```

Betyder det at brugeren af EntryEx også må få adgang til public variable og funktioner i Entry, hvis vi skrev private var det kun EntryEx der måtte tilgå variable og funktioner i Entry.

Den findes en udgave mere som hedder protected. Den betyder at det kun er class'en selv og class'er der arver fra clas'en der må få adgang.

Dette kan synes et anelse besværligt og overflødigt. Men hvis du tænker på en class som std::ofstream må den have en mængde interne variable og funktioner til at holde styr på filen, dens størrelse, placering på harddisken osv. Hvis du kunne få adgang til disse kunne du komme til at lave en mængde ting som kunne have fatale konsekvenser for din harddisk.

Opgave:

Lav det nyeste regnemaskine eksempel om så alle class'ens variable er private og lav en nogle public funktioner til at tilgå disse med.

Afsnit 27

Alle artikler skal have en ende, denne artikel ender her. Dette betyder ikke at du nu har lært alt om C++, for det har du ikke. Der er skrevet adskillige bøger på mange hundrede sider om C++

Men jeg håber at du har forstået lidt af hvordan det fungerer, og har fået lyst til at fortsætte.

Du skal lige have en sidste information. Ofte er det praktisk at kunne skrive kommentarer i koden, som ikke skal være en del af programmet, men som kan hjælpe den der læser koden. Det kunne være en beskrivelse af hvad koden, funktionen eller programmet gør, eller ting som er vigtig at huske for at forstå hvordan programmet virker.

Der er to måder at lave kommentarer på. Den ene starter med // og gælder resten af linjen, den anden består af alt mellem /* og */, og kan dække flere linjer. Nogle eksempler:

```
/* Dette program skrive 123 til test.txt
** Det er ikke specielt brugbart, men en text
*/
// Starter med include filer, alle standard
#include <fstream>
#include <iostream>
int main()
{  /* Her starter programmet */
   std::ofstream Fil("test.txt");
   if(!Fil)
   {  // Hvis ikke filen kunne åbnes
```

```
        std::cout << "Kunne ikke åbne test.txt" << std::endl;
        return 1;
    }
    Fil << 123;
    // Fil bliver lukket af systemet når vi returnerer fra main
}
```

Du vil sikkert opdage at programmet skriver kommentarer i en anden farve, for at markere at det netop er kommentarer.