Norges teknisk–naturvitenskapelige universitet
Institutt for datateknikk og informasjonsvitenskap

TDT4102 Prosedyre og Objektorientert programmering Vår 2015

Øving 8

Frist: 2015-03-20

Aktuelle temaer for denne øvinga:

- Klasser, arv, polymorfi og virtuelle funksjoner
- Unsigned og const

Generelle krav:

- Bruk de eksakte navn og spesifikasjoner som er gitt i oppgava.
- Det er valgfritt om du vil bruke en IDE (Visual Studio, XCode), men koden må være enkel å lese, kompilere og kjøre.

Anbefalt lesestoff:

- Kapittel 14 og 15, Absolute C++ (Walter Savitch)
- It's Learning-notater

NOTAT

- Alle **uthevede** spørsmål skal besvares.
- Eventuelle andre spørsmål i teksten er ment som refleksjonsoppgaver, men anbefales å gjøres da de er en del av øvinga og øvingsopplegget er eksamensrelevant.
- Forsikre deg om at du sammen med øvingsoppgava fikk utdelt Image.h og ImageIO.cpp

Introduksjon:

Denne øvingen har to deler, som begge omhandler temaet arv. I den første delen skal vi implementere en vektorklasse ved å spesialisere matriseklassen fra øving 6. I den andre delen av øvingen skal du implementere en bildeklasse som representerer en Bitmap Image File, og så skal du tegne geometriske figurer til denne filen.

Utdelte filer:

Disse filene skal lastes ned, lagres sammen med de andre .h og .cpp i øvinga og legges til prosjektet.

For vektoroppgaven kan du selv velge om du vil benytte seg av din egen matriseklasse fra øving 6, eller om du har lyst å bruke koden fra løsningsforslaget (som også finnes i den utdelte kodem).

For bildebehandlingsdelen skal øvingen løses uten å gjøre endringer i de utdelte filene (med unntak av Image.cpp), og det forventes at du har satt deg inn i koden i filen `Image.h`.

- `Matrix.h`: Deklarasjonen til matriseklassen fra øving 6.
- `Matrix.cpp`: Implementasjonen av matriseklassen fra øving 6.
- `Image.h`: Deklarasjonen til bildeklassen du skal implementere. Ikke endre, men **ta en titt på denne.**
- `Image.cpp`: Denne filen inneholder tomme funksjoner for koden du skal implementere i oppgave 4.
- `ImageIO.cpp`: Inneholder funksjonalitet for å lagre Image-objekter til bmp-filer på disk.

Andre relevante kilder:

- RGB fargemodellen - Wikipedia
  `http://en.wikipedia.org/wiki/RGB`.
- Hvordan representere et todimensjonalt array med en endimensjonal liste. Se forelesningsnotater.

1 Implementere Vector ved å arve Matrix

I lineær algebra er det mange likhetstrekk mellom matriser og vektorer, for eksmpel kan man behandle en vektor som en matrise med en enkelt rad eller kolonne. Dette lar oss definere matrise-produktet mellom matriser og vektorer som følger (Vi bryr oss ikke om rad-vektorer enn så lenge.):

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} * \begin{bmatrix} e \\ f \end{bmatrix} = \begin{bmatrix} a*e+b*f \\ c*e+d*f \end{bmatrix}$$

a) Lag en klasse `Vector` som arver `Matrix`-klassen.

Vi vil bruke Vector-klassen vår som et grensesnitt for en Mx1-dimensjonal matrise med noe ekstra funksjonalitet (altså en kolonnevektor).

NB: Det er her snakk om en matematisk vektor, og akkurat som i tidligere øvinger **ikke std::vector** fra standardbiblioteket til C++.

b) Implementer følgende konstruktører for Vector-klassen.

`Vector()`

- Standardkonstruktør, skal initalisere den underliggende matrisen til en ugyldig tilstand.

```
explicit Vector(unsigned int nRows)
```

- Skal konstruere den underliggende nRows x 1-matrisen, initalisert som 0-matrisen (explicit-nøkkelordet er ikke en del av pensum, men det bør brukes her.)

```
Vector(const Matrix & other);
```

- Konverterings-konstruktør for Matrix. Skal tilordne en matrise til `*this` hvis og bare hvis matrisen har dimensjoner Nx1, ellers skal resultatet `*this` settes til invalid.
  Hint: Dette kan løses på to måter ved å gjenbruke funksjonalitet du allerede har skrevet. 1 er enklest å forstå, 2 er mest effektiv.
  1. Du kan enten bruke copy-konstruktøren til for å lage en kopi av høyresiden, og deretter invalidere matrisen hvis du resultatet er invalid. (Dette krever at du har en `invalidate()` funksjon i `Matrix` klassen)
  2. Eller, du kan gjenbruke operator= fra matriseklassen. Bruk først den tomme konstruktøren fra matrix, for å garantere at Vektoren er invalid. Kall deretter matriseklassen sin `operator=` direkte ved å skrive `Matrix::operator=(other);` hvis matrisen har gyldige dimensjoner.

Faktisk trenger vi ikke implementere destruktøren for Vector-klassen. Siden vektoren ikke har noen egne medlemsvariabler, vil kompilatoren lage en destruktør som kaller baseklassens destruktør, som dermed destruerer objektet korrekt.

Hint: Bruk en initsialiseringsliste for å kalle Matrix sin konstruktør. Legg merke til at vi slipper å implementere en egen kopikonstruktør og operator= for Vector fordi den har ingen egne dynamisk allokerte medlemmer. Kompilatoren generer standardversjoner som gjenbruker vår implementasjon for Matrix-delen av Vector.

c) Implementer get- og set-funksjoner for Vector

Det gir ikke mening å snakke om kolonner i en Vector, siden den bare har én. Derfor bør vi implementere nye get- og set-funksjoner som bare tar inn rad som parameter.

Oppgaven er dermed å **implemetere følgende to funksjoner:**

```
void set(unsigned int row, double value)
```

- Setter verdien på rad `row` i vektoren til `value`.

```
double get(unsigned int row) const
```

- Returnerer verdien på rad `row` i vektoren.

NB: Når vi implementerer disse get- og set-funksjonene overskygger vi faktisk get- og set-funksjonene i den opprinnelige klassen, og gjør dem usynlige gjennom vektor-grensesnittet.

Hint: Du kan eksplisitt kalle base-klassens utgave av get og set inni klassen Vector ved å skrive

`Matrix::get(r, c)`

og

`Matrix::set(r, c, v).`

(Der r, c og v erstattes med passende verdier.)

d) Implementer følgende medlemsfunksjoner:

```
double dot(const Vector &rhs) const
```

- Returnerer dot-produktet mellom `this` og `rhs`. Hvis vektorene har forskjellig dimensjon, eller er ugyldig, returner `nan("");`. Denne finner du i `<cmath>`-headeren.

```
double lengthSquared() const
```

- Returnerer den kvadrerte lengden av vektoren.

```
double length() const
```

- Returnerer lengden av vektoren. Kvadratorota av en verdi kan beregnes med funksjonen `sqrt(double)` som finnes i `<cmath>`-headeren.

Hint: Du kan gjenbruke dot-funksjonen for å beregne lengthSquared og gjenbruke lengthSquared for å beregne length().

2 Testing og forståelse

På dette stadiet i forrige matriseøving trengte vi å implementere alle operatorene (+, -, *, +=, -=, *=) for Vector-klassen, i tillegg til operatorer for Matrix-Vector-samhandling. Denne gangen trenger vi ikke det. Funksjonaliteten er allerede implementert i Matrix-klassen.

Så, hvordan har dette skjedd?

Som du leste tidligere i denne øvinga, er en matrise-vektor-multiplikasjon, det samme som å venstre-multiplisere en matrise med en kolonne vektor. I tillegg har du i denne øvinga latt Vector arve alle egenskapene ved Matrix, (alt bortsett fra konstruktørene og operator=). Dermed vil vektoren bli behandlet som en instans av base-klassen (i dette tilfellet en Mx1 Matrix) hver gang du bruker operator* mellom en matrise og en vektor. Operandene blir så multiplisert etter reglene satt i Matrix::operator*, som resulterer i en Mx1 Matrix, (eller en ugyldig matrise, dersom dimensjonene ikke passer overens.)

Dette er et eksempel på bruk av arv for å redusere programmererens arbeidsmengde betraktelig. Ta en titt på mengden kode som trengs for denne vektorklassen, sammenlignet med klassen du skrev i øving 5 (eventuelt LF fra øving 5). Forskjellen er betydelig. Dette er styrken ved objektorientert programmering.

a) Test implementasjonen din, lag en 4x4 ikke-identitetsmatrise, og multipliser den med en ikke-0 vektor. Sjekk resultatene for hånd.

3 Introduksjon til bildebehandlingsoppgaven

Denne oppgava kan hoppes over hvis du er kjent med begrepene «piksel», «RGB», «bitmap» og «unsigned» datatyper. Er du i tvil om du er kjent med disse begrepene, gjør oppgava!

a) Bilder kan representeres digitalt som et nett av punkter, hvor hvert punkt har en bestemt farge. I datamaskinverdenen kalles disse punktene for «piksler», og størrelsen på bilder blir som regel oppgitt på formen X×Y, der X er bildets bredde og Y er dets høyde i piksler. Disse verdiene omtales som bildets oppløsning («dimensjoner» er også brukt, dog mindre ofte) og er angitt i piksler. Altså har et bilde med en oppløsning på 1366×768 1366 piksler i bredden og 768 i høyden.

Hva er dimensjonene til dette bildet `http://www-ee.uta.edu/dip/courses/ee5356/lena512.bmp`?

b) I digitale bilder har hver av pikslene en gitt farge. For å kunne representere en farge digitalt trenger vi en fargemodell, som for eksempel RGB. RGB er en fargemodell hvor fargene **R**ød, **G**rønn og **B**lå blandes sammen i forskjellige forhold for å lage andre, mer spennende, farger. I motsetning til den subtraktive fargemodellen vi lærte om på barneskolen (hvor rød, gul og blå var primærfargene) er RGB-modellen additiv. I en additiv fargemodell får man hvitt hvis man blander alle primærfargene; i en subtraktiv fargemodell får man sort ved å blande alle primærfargene.

Hvilken farge får man hvis man ikke blander sammen noen farger i en additiv fargemodell?

c) For å representere en farge i RGB-modellen (Rød, Grønn og Blå) trengs det en separat verdi for hver av fargene. Altså en verdi for å indikere hvor mye rød vi skal ha, en annen verdi som indikerer hvor mye grønt og en tredje verdi for å indikere mengden blått. Det er vanlig å bruke verdier mellom 0 og 255 for å indikere mengden av hver farge. For eksempel: «R=0, G=0, B=255» er en måte vi kan definere en farge i RGB-modellen (i dette tilfelle er fargen «blå», eller «det blir ikke mer blå enn dette»). Vi kan også uttrykke dette på formen «0, 0, 255» eller hexadesimalt med «0x0000FF». Uansett så er den første verdien rød, den andre verdien grønn og den siste verdien blå.

Hvordan kan vi representere rød i RGB-modellen?

d) Å representere en farge i RGB-modellen krever altså tre verdier. I C++ kan vi for eksempel bruke tre variabler, en for hver farge. I `Image.h` finnes det en struct ved navn «Color» som gjør akkurat dette: den har tre `unsigned char`-variabler som representerer mengden av hver primærfarge i fargen. Unsigned er engelsk for «uten fortegn». En vanlig (`signed`) `char` kan lagre verdier mellom -128 og 127 (husk at bokstaver egentlig bare er tall). Unsigned char kan lagre verdier mellom 0 og 255, noe som passer ypperlig til vårt formål da hver fargeverdi i RGB-modellen vår kan ha verdier fra og med 0, til og med 255!

Hvorfor bruker vi `unsigned char` til å lagre fargedata i bildeklassen?

Hvorfor kan det være lurt å bruke unsigned variabler til å lagre høyden og bredden til bildeklassen?

4 Grunnarbeidet - Image klassen

Du har til denne oppgava fått utdelt noen filer, blant annet fila `Image.h`, som inneholder klassespesifikasjonen til Image-klassen. Sørg for at denne, og `ImageIO.cpp` er del av prosjektet ditt. Implementasjonene du skal gjøre i denne oppgava skal gjøres i `Image.cpp`, som du må opprette selv. Du slipper altså å skrive klassedeklarasjonen for Image-klassen.

a) Implementer konstruktøren for `Image`-klassen. Husk at konstruktører brukes til å sette opp objektet før bruk, og den må derfor initialisere alle medlemsvariablene. Arrayen som holder pikseldataene må allokeres dynamisk.

1. Det trengs 3 `unsigned char` for å lagre fargen til en piksel. Hvor mange `unsigned char`s trenger man for et helt bilde? *Hint: se Figur 1.*

$R_{(0,0)}$	$G_{(0,0)}$	$B_{(0,0)}$	$R_{(1,0)}$	$G_{(1,0)}$	$B_{(1,0)}$	$R_{(0,1)}$	$G_{(0,1)}$	$B_{(0,1)}$	$R_{(1,1)}$	$G_{(1,1)}$	$B_{(1,1)}$

Figur 1: Et eksempel på hvordan fargedataene til et 2×2 bilde kan representeres i en tabell.

2. *Hint: Selv om bilder gjerne har to dimensjoner, kan vi representere det med et en-dimensjonalt array gitt at det har konstant bredde, se Figur 2 og 3.*

6	7	8	9	10	11
0	1	2	3	4	5

Figur 2: En to-dimensjonalt tabell med plass til 12 elementer.

0	1	2	3	4	5	6	7	8	9	10	11

Figur 3: Den samme tabellen som i Figur 2 presentert som et en-dimensjonalt array.

b) Implementer destruktøren for Image-klassen.

c) Skriv medlemsfunksjonene `getWidth()` og `getHeight()` for `Image`-klassen.

1. Hva betyr `const` som står etter disse funksjonene?
2. Hvorfor burde vi bruke `const` her?

d) Skriv `getScanLine()`-medlemsfunksjonene.
Disse skal begge returnere en peker til første piksel i den valgte linjen i bildets dataområde (en såkalt *scanline*). Merk at `getScanLine()` brukes av `saveImage()` og `loadImage()` funksjonene og må derfor være skrevet korrekt før vi kan lagre bilder til fil.

1. Én av disse er `const`, den andre er ikke. Hvorfor?

e) Implementer `setColor()`-medlemsfunksjonen. Denne skal sette fargeverdien til én piksel i bildet. Husk at fargeverdien består av tre forskjellige komponenter som skal lagres etter hverandre i bildedataene.

1. Pass på at `setColor()` ikke prøver å endre på punkter utenfor bildet!
2. Hvordan får man tak i array-indeksen til pikselen på posisjon (x, y) i bildetabellen?

NB! Merk at i et BMP bilde er origo i nedre venstre hjørne, akkurat som i et kartesisk koordinatsystem.

f) Skriv `getColor()`-medlemsfunksjonen. Denne skal returnere fargeverdien til en piksel i bildet.

1. Test også at `getColor()` returnerer den samme fargen som ble satt av `setColor()`

g) Skriv `fill()`-funksjonen. Denne skal fylle hele bildet med samme farge.

h) Bruk `saveImage()`-funksjonen, deklarert i `Image.h`, til å lagre et bilde med bakgrunnsfargen R=193, G=84, B=193. Denne fargen kalles *crayola fuchsia*. Vi sa RGB kunne brukes til å lage mange spennende farger!

1. Eksperimenter selv med andre, spennende fargekombinasjoner. Bare fantasien setter grenser!
2. Bruk også `setColor()` til å sette ett rødt punkt på en hvit bakgrunn. Hvis punktet tegnes på feil sted har du en feil i `setColor()`.

5 Abstrakte klasser - `Shape`-klassen

Du kan nå fylle et bilde med farger og sette enkeltpunkter til andre farger; nå skal du lage et sett med klasser som representerer forskjellige geometriske figurer som kan tegnes inn i et bilde. Siden C++ er et objektorientert språk skal vi løse dette på en objektorientert måte ved å lage en klasse som inneholder fellestrekkene til de geometriske figurene vi har lyst til å kunne tegne.

a) Opprett klassen `Shape`

b) `Shape` skal ha en medlemsvariabel for å lagre farge. Denne variabelen skal være privat, og skal settes i konstruktøren. **Legg til denne medlemsvariabelen.**

Tips: Bruk `Color`-structen fra `Image.h`.

1. Hvordan kan vi hente ut fargen til en `Shape` når medlemsvariablen er private?

c) `Shape` skal ha en pure virtual medlemsfunksjon kalt `draw()`; denne funksjonen skal ta inn en referanse til et `Image`-objekt som argument. I form-klassene som skal lages etterpå skal denne funksjonen implementeres og brukes til å tegne formen på et bilde. **Legg til medlemsfunksjonen `draw()`.**

1. Hva vil det si at en funksjon er pure virtual?
2. En klasse som inneholder en pure virtual-medlemsfunksjon kalles en abstrakt klasse. Hva kan vi ikke gjøre med abstrakte klasser?

6 Spesialisering av baseklasser - `Line`-klassen

Den første figuren vi skal lage er en av de aller eldste geometriske formene, nemlig linjen. Eller, da en linje strengt tatt er uendelig i lengde er det linjestykker vi skal tegne. Et linjestykke kan defineres ut i fra et startpunkt og et endepunkt. I denne oppgava kan du gå ut fra at startpunktet alltid vil ligge til venstre for endepunktet. (Hvis du vil ha en ekstra utfordring, kan du gå ut fra at dette ikke alltid er tilfellet, og skrive koden din deretter. Dette er imidlertid ikke påkrevd.)

a) Lag klassen `Line`. Denne skal arve `Shape`.

1. Hvilke argumenter trenger konstruktøren til en linjeklasse å ta inn? Husk at alle figurer arver fargen fra `Shape`!
2. Hvordan kaller man konstruktøren til `Shape` fra konstruktøren til `Line`?

b) Implementer `draw()`-funksjonen for `Line`-klassen.

1. Hvordan tegner man en linje? Her må du blåse støvet av matte 1-boken din og finne det ut! Alternativt kan du bruke topunktsformelen fra ungdomskolen:

$$y = \frac{y_2 - y_1}{x_2 - x_1}(x - x_1) + y_1$$

.

NB! Pass på at du får korrekt oppførsel mtp. heltallsdivisjon i formelen over.

c) Test klassen din ved å tegne linje med startpunkt (0, 0) og sluttpunkt (100, 50). Lagre bildet for å se resultatet.

d) Test klassen din igjen ved å tegne en linje med startpunkt (0, 0) og sluttpunkt (100, 200).

1. Ble denne linjen like fin som den forrige? Hvis du har fulgt oppgaveteksten er svaret nei. Årsaken til dette er fordi implementasjonen av formelen $y = xm + a$ tegner linjen mens den beveger seg langs x-aksen. Dette fungerer helt fint for forholdsvis horisontale linjer, men fører til krøll hvis linjen er for bratt (altså hvis stigningstallet til linjen er større enn 1).

e) Skriv om `draw()`-funksjonen til `Line`-klassen slik at den sjekker stigningstallet til linjen før den begynner å tegne. Hvis stigningstallet er større enn 1 skal `draw()`-funksjonen bevege seg langs y-aksen.

1. *Hint: Vi kan skrive om topunktsformelen til å gi x som en funksjon av y:*

$$x = \frac{x_2 - x_1}{y_2 - y_1}(y - y_1) + x_1$$

7 Flere spesialiseringer - `Rectangle` klassen

Linjer kan brukes til å lage rektangler. Hvordan, lurer du? Det er slik at et rektangel har fire sider og fire hjørner, men kan defineres ut i fra dets diagonal, som kan defineres som en linje! Husk at et rektangel har fire sider som er parvis like.

a) Lag klassen `Rectangle`. Denne skal, i likhet med `Line`, arve `Shape`.

1. Hvilke argumenter trenger `Rectangle`-klassen sin konstruktør å ta inn?

b) Skriv `draw()`-funksjonen for klassen `Rectangle`.

1. *Hint: Prøv å tegne diagonalen først.*

8 Enda flere spesialiseringer - `Circle` klassen

Rette linjer er i grunnen ganske kjedelig, synes du ikke? Vel hold deg fast for nå skal vi tegne sirkler, geometriens rareste objekt. En sirkel er definert ved et midtpunkt og en radius. Alle punktene langs en sirkels periferi er like langt fra dets midtpunkt.

a) Lag klassen `Circle`. I likhet med de fleste andre klassene i denne øvinga skal den arve `Shape`.

1. Hvilke argumenter trenger `Circle`-klassen sin konstruktør å ta inn?

b) Skriv `draw()`-funksjonen for klassen `Circle`. Formelen for punktene i en sirkel med radius r og sentrum i (x_0, y_0)er:

$$(x - x_0)^2 + (y - y_0)^2 = r^2$$

1. *Hint: her kan det være en idé å iterere over både x- og y-aksen.*

9 Polymorfi med virtuelle funksjoner

For å lage fantastiske bilder kan det være greit å kunne samle flere forskjellige figurer på et «lerret». Du får nå se hvor praktisk det er når alle klassene arver samme klasse.

a) Skriv klassen `Canvas`. Denne klassen skal inneholde en `std::vector` av `Shape`-pekere.

b) Skriv medlemsfunksjonen `addShape()` som tar inn en `Shape`-peker og legger den til i vektoren.

c) Skriv medlemsfunksjonen `rasterizeTo()` som skal ta inn en referanse til et `Image`-objekt. Funksjonen skal tegne alle `Shape`-objektene i vektoren inn i bildet.

d) Bruk `Canvas`-objektet ditt til å tegne et smilefjes. Bruk linje-segmenter til å lage munnen, eller (valgfritt) skriv en ny klasse for å tegne halvsirkler.

e) *Valgfri: Bruk `loadImage()`-funksjonen til å laste inn bildet referert til i Oppgave 1a. Du må først lagre det på harddisken. Bruk `Canvas`-klassen til å tegne en bart på damen.*