Norges teknisk–naturvitenskapelige universitet
Institutt for datateknikk og informasjonsvitenskap

TDT4102 Prosedyre- og objektorientert programmering Vår 2017

Øving 10

Frist: 2016-04-08

Aktuelle temaer for denne øvingen:

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

Generelle krav:

- Bruk de eksakte navn og spesifikasjoner som er gitt i oppgaven.
- 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)

Bakgrunn for oppgavene

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 7. I den andre delen av øvingen skal du implementere en bildeklasse som representerer en BMP-fil (også kjent som en *bitmap*-fil), og deretter tegne geometriske figurer.

Utdelte filer

Last ned disse filene fra It's learning (`oving10.zip`), og legg dem til i prosjektet ditt.

For vektoroppgaven kan du selv velge om du vil benytte seg av din egen matriseklasse fra øving 7, eller om du har lyst å bruke koden fra løsningsforslaget (som også finnes i den utdelte koden).

For bildebehandlingsdelen skal øvingen løses uten å gjøre endringer i de utdelte filene (med unntak av `Image.cpp`), men sett deg inn i koden fra filen `Image.h` før du begynner på bildebehandlingsoppgavene.

- `Matrix.h`: Deklarasjonen til matriseklassen fra øving 7.
- `Matrix.cpp`: Implementasjonen av matriseklassen fra øving 7.
- `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 2.
- `ImageIO.cpp`: Inneholder funksjonalitet for å lagre Image-objekter til bmp-filer på disk.

Introduksjon til bildebehandling

Bilder kan representeres digitalt som et nett av punkter, der hvert punkt har en bestemt farge. I datamaskinverdenen kalles disse punktene for *piksler*. Størrelsen på bilder blir som regel oppgitt på formen $X \times Y$, der X er bildets bredde og Y er dets høyde i piksler. Dette kalles ofte bildets *oppløsning* eller *dimensjoner*. Et bilde med en oppløsning på 1366×768 har altså 1366 piksler i bredden og 768 i høyden.

I et digitalt bilde har hver av pikslene en gitt farge. For å kunne representere en farge digitalt trenger vi en *fargemodell* som beskriver fargen. Den vanligst brukte fargemodellen i datasammenheng er RGB, hvor fargene **r**ød, **g**rønn og **b**lå blandes sammen i forskjellige forhold for å lage andre 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, mens man i en subtraktiv fargemodell får svart ved å blande alle primærfargene.

For å representere en farge i RGB-modellen bruker vi en separat verdi for hver av de tre fargene. Vi har altså én verdi for å indikere hvor mye rødt fargen skal inneholde, én verdi som indikerer hvor mye grønt den skal inneholde og én verdi for å indikere mengden blått. Det er vanlig å bruke verdier mellom 0 og 255 for å indikere mengden av hver farge. Vi kan for eksempel skrive `R=0, G=0, B=255` for å definere fargen «blå» (eller «det blir ikke mer blå enn dette») i RGB-modellen. Dette kan også uttrykkes på formen $(0, 0, 255)$ eller heksadesimalt som `0x0000FF`. I alle tilfeller representerer uansett

den første verdien mengden av rød, den andre verdien mengden av grønn og den siste verdien mengden av blå.

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 variabler av typen `unsigned char`, 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 `char`-variabler i C++ egentlig bare er tall). `unsigned char` kan på sin side lagre verdier mellom 0 og 255. Dette passer ypperlig til vårt formål da hver farge i RGB-modellen har en verdi mellom 0 og 255.

For å sjekke om du forstår de grunnleggende bildebehandlingsbegrepene vi introduserte her, kan du stille deg selv følgende spørsmål:

- Hva er dimensjonene til dette bildet?
- Hvilken farge representerer $(0, 0, 0)$ i RGB-modellen?
- Hvordan representerer vi rødfarge i RGB-modellen?
- Hvorfor ville vi brukt `unsigned char` til å lagre fargedata i en bildeklasse?

Annet relevant lesestoff

- RGB-fargemodellen - Wikipedia
- Hvordan representere et todimensjonalt array med en endimensjonal array. Se forelesningsnotater.

1 Implementere `Vector` ved å arve `Matrix` (25%)

Dersom vi skulle lagd en `Vector`-klasse i øving 7 måtte vi implementert alle operatorene vi ville bruke for hånd for klassen, altså `+`, `-`, `*`, `+=`, `-=`, `*=` og så videre, i tillegg til all funksjonalitet for samhandling mellom `Vector` og `Matrix`. Det er imidlertid mange likhetstrekk mellom matriser og vektorer, og dette gjør at vi kan behandle en vektor som en matrise med én rad (radvektor) eller én kolonne (kolonnevektor) og spare oss selv for mye arbeid. I denne oppgaven skal vi se på hvordan vi ved hjelp av arv kan bruke `Matrix`-klassen til å lage en klasse for representasjon av kolonnvektorer. På den måten kan vi automatisk gi klassen alle egenskapene til `Matrix` uten å måtte implementere disse manuelt.

Dersom vi for eksempel ønsker å multiplisere en vektor med en matrise kan vi se på vektoren som en matrise med størrelse $M \times 1$, kan vi bruke `Matrix::operator *` til å multiplisere vektoren med matrisen og få samme resultat.

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

Ved å gjøre dette kan vi spare oss for å måtte skrive mange nye linjer kode for håndtering av vektorer. Ikke bare reduserer vi programmererens arbeidsmengde betraktelig, men ved kun å ha funksjonaliteten ett sted i koden reduserer vi sjansen for at det oppstår feil. Dette illustrerer styrken ved objektorientert programmering.

a) Lag en klasse `Vector` som arver `Matrix`-klassen. I stedet for å lage `Vector` helt fra bunnen av, ønsker vi å se på vektorer som Mx1-dimensjonale matriser. Dette lar oss automatisk bruke all funksjonaliteten som ligger i `Matrix`-klassen, og vi kan i tillegg legge til litt ekstra funksjonalitet som kun skal gjelde for vektorer.

b) Implementer følgende konstruktører for Vector-klassen. Du bør her bruke initialiseringslister for å kalle `Matrix` sine konstruktører.

`Vector()`

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

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

- Skal konstruere den underliggende nRows x 1-matrisen og initalisere den som 0-matrisen. `explicit`-nøkkelordet er ikke en del av pensum, men det bør brukes her. Nøkkelordet forhindrer at konstruktøren brukes til å automatisk konvertere verdier. Dersom vi her *ikke* hadde brukt `explicit`, kunne vi puttet en `int` inn i en hvilken som helst funksjon som tar inn en `Vector`. `int`-verdien ville da bli konvertert til en `Vector` ved hjelp av konstruktøren. Det vil vi unngå her siden det lett kan føre til feil.

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

- Konverteringskonstruktør fra `Matrix` til `Vector`. En konverteringskonstruktør er en konstruktør som tar inn én parameter av en annen type, og ikke er deklarert med `explicit`-nøkkelordet. Skal tilordne matrisen `other` til `*this` hvis og bare hvis matrisen har dimensjoner Nx1, ellers skal resultatet `*this` settes til invalid.
  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 for å lage en kopi av høyresiden, og deretter invalidere matrisen hvis resultatet er invalid. (Dette krever at du har en `invalidate()` funksjon i `Matrix` klassen)
  2. Alternativt kan du gjenbruke `operator =` fra `Matrix`. Bruk først den tomme konstruktøren fra `Matrix` for å garantere at vektoren er ugyldig. Kall deretter `Matrix` sin `operator =` direkte ved å skrive `Matrix::operator =(other);` dersom matrisen har gyldige dimensjoner.

Vi slipper å implementere en egen destruktør, kopikonstruktør og `operator =` for `Vector` fordi den ikke har noen egne medlemsvariabler. Kompilatoren generer standardversjoner som gjenbruker implementasjonene i base-klassen `Matrix`.

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 egne `get-` og `set`-funksjoner i `Vector` som bare tar inn rad som parameter.

Oppgaven er dermed å implementere følgende to funksjoner:

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

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

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

- Returnerer verdien på rad `row` i vektoren.

Merk at når vi implementerer disse `get-` og `set`-funksjonene «overskygger» vi `get-` og `set`-funksjonene i `Matrix`-klassen, og gjør dem usynlige når `Vector` brukes *selv om* klassen arver fra `Matrix`.

Hint: Du kan eksplisitt kalle base-klassens utgaver 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 av passende verdier.

d) Implementer følgende medlemsfunksjoner:

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

- Skal returnere prikkproduktet (på engelsk «dot product») mellom `this` og `rhs`. Hvis vektorene har forskjellig dimensjon, eller er ugyldige, skal du returnere `nan("");`. Denne funksjonen finner du i `<cmath>`.

```
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().

e) Test implementasjonen din. Lag en 4×4 ikke-identitetsmatrise, og multipliser den med en ikke-0 vektor. Bruk for eksempel tallene nedenfor.

$$\begin{bmatrix} 8 & 8 & 8 \\ 8 & 8 & 8 \\ 8 & 8 & 8 \end{bmatrix} * \begin{bmatrix} 5 \\ 5 \\ 5 \end{bmatrix} = \begin{bmatrix} 120 \\ 120 \\ 120 \end{bmatrix}$$

2 Image-klassen (15%)

Du har til denne oppgaven fått utdelt noen filer, blant annet filen `Image.h`, som inneholder deklarasjonen for `Image`-klassen. Sørg for at både denne og `ImageIO.cpp` er del av prosjektet ditt. I denne oppgaven skal du skrive all koden din i `Image.cpp`, og denne må du opprette selv. Husk å følge spesifikasjonene gitt i `Image.h`.

Før du begynner på denne oppgaven anbefales det at du leser over «Introduksjon til bildebehandling» i starten av øvingen.

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

1. I headerfilen `Image.h` finner du `typedef Color Pixel;` som sier at Pixel er et alias for Color (altså at de betyr det samme).
2. Det trengs 3 `unsigned char` for å lagre fargen til en piksel. Hvor mange `unsigned chars` trenger man for et helt bilde? 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.

3. Selv om et bilde gjerne har to dimensjoner, kan vi representere det med et en-dimensjonalt array gitt at det har konstant bredde. Se Figur 2 og 3. Merk at vi i denne oppgaven jobber med BMP-bilder, som har origo i nedre venstre hjørne på samme måte som i et kartesisk koordinatsystem.

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

Figur 2: En to-dimensjonal 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. Hva betyr `const` som står etter disse funksjonene?

d) Skriv de to `getScanLine()`-medlemsfunksjonene. Disse skal begge returnere en peker til første piksel i den valgte linjen i bildets dataområde (en såkalt *scanline*). Én av de to funksjonene er `const`, den andre ikke. Hvorfor?

e) Implementer `setPixelColor()`-medlemsfunksjonen. Denne skal sette fargeverdien til én piksel i bildet.

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

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

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

g) Skriv `fill()`-medlemsfunksjonen. 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 forøvrig «crayola fuchsia». Bruk også `setPixelColor()` til å sette ett rødt punkt på en hvit bakgrunn. Hvis punktet tegnes på feil sted har du en feil i `setPixelColor()`.

3 Abstrakte klasser: `Shape`-klassen (15%)

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`. Klassen skal ha:

- En medlemsvariabel for å lagre farge. Bruk `Color`-structen fra `Image.h`. Denne variabelen skal være `private`.
- En konstruktør som tar inn og setter fargen.
- En `get`-funksjon, `getColor`, for å kunne hente ut verdien til den `private` medlemsvariabelen som inneholder fargen.

b) `Shape` skal ha en pure `virtual` medlemsfunksjon kalt `draw()`. Denne funksjonen skal ta inn en referanse til et `Image`-objekt som parameter. I form-klassene vi skal lage 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?

4 Spesialisering av baseklasser: `Line`-klassen (15%)

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 oppgaven 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 kan du bruke topunktsformelen du lærte på ungdomskolen. Pass på at du får korrekt oppførsel mtp. heltallsdivisjon i formelen.

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

c) Test klassen din ved å tegne en 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).

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

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

5 Mer spesialisering: `Rectangle`- og `Circle`-klassene (15%)

Linjer kan brukes til å lage rektangler. Et rektangel har fire sider og fire hjørner, men hvis vi antar at sidene er parallelle med x- og y-aksene kan et rektangel også defineres ut i fra dets diagonal, som selvfølgelig er en linje.

a) **Lag klassen `Rectangle`.** Denne skal, i likhet med `Line`, arve `Shape`. Hvilke parametre trenger `Rectangle`-klassen sin konstruktør å ta inn?

b) **Skriv `draw()`-funksjonen for klassen `Rectangle`.**
Hint: Prøv å tegne diagonalen først.

(Valgfritt) Vi skal nå se på noe litt mer interessant, nemlig tegning av sirkler. En sirkel er definert ved et midtpunkt og en radius. Alle punktene langs en sirkels periferi er like langt fra dets midtpunkt.

c) **(Valgfritt) Lag klassen `Circle`.** Klassen skal arve `Shape`. Hvilke parametre trenger `Circle`-klassen sin konstruktør å ta inn?

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

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

Hint: Her kan det være en idé å iterere over både x- og y-aksene.

6 Polymorfi med virtuelle funksjoner (15%)

For å lage mer interessante bilder kan det være greit å kunne samle flere forskjellige figurer på et «lerret». Du får nå se hvor praktisk det er at alle klassene vi har skrevet arver samme klasse.

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

b) **Skriv medlemsfunksjonen `addShape()`.** Denne skal ta inn en `Shape`-peker og legger den til i vektoren.

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

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

e) **(Valgfritt)** Bruk `loadImage()`-funksjonen til å laste inn dette bildet (du kan lese mer om bildet her). Du må først lagre det på harddisken. **Bruk `Canvas`-klassen til å tegne en bart på kvinnen i bildet.**