Norges teknisk–naturvitenskapelige universitet
Institutt for datateknikk og informasjonsvitenskap

TDT4102 Prosedyre og Objektorientert programmering Vår 2014

Øving 5

Frist: 2014-02-21

Mål for denne øvinga:

- Lære om operatorar og interaksjon mellom klasser av forskjellige typer.
- Lære å implementere og bruke klasser.

Generelle krav:

- Bruk dei eksakte namn og spesifikasjonar som er gitt i oppgåva
- Det er valfritt om du vil bruke ein IDE (Visual Studio, Xcode), men koden må vere enkel å lese, kompilere og kjøre.

Anbefalt lesestoff:

- Kapittel 5.4, 7, 8.1, 8.3,& 10.3 Absolute C++ (Walter Savitch)

Merk: Du skal i denne øvinga, ved å bruke klasser, definere dine eigene typer. Når du gir namn til typane dine, skal du bruke store/små bokstavar på samme vis som når du gir namn til variablar, men med den forskjellen at første bokstav i namnet skal vere stor. Dette er kun ein konvensjon for at det skal vere lettare å lese koden. Når ein implementerer klasser er det óg vanleg å lage seg **ei** .h-fil og **ei** .cpp-fil per klasse (t.d. om klassa heiter «Car», lager vi Car.cpp og Car.h, og skriv all koden for klassa «Car» i disse filane.) Følg denne norma i oppgåver der det bes om å skrive klasser.

Typenamna dine skal difor vere på forma

```
MyType
```

mens variablar, som du sikkert husker, ser ut som

```
myVariable
```

og til slutt konstantar

```
MYCONSTANT
```

1 Ei enkel 2x2 Matriseklasse(20%)

Ei matrise er eit matematisk konstrukt frå lineær algebra, og nyttast i fleire sidar ved vitskap, mellom anna datateknikk og programmering.

Reint praktisk er ei matrise eit strukturert rutenett av tal som har eit sett reglar om korleis matriser samhandlar med kvarandre. I denne delen av øvinga skal vi ta ein kikk på den enklaste formen for matrisar, 2x2-matrisar. I dei påfølgande delane skal vi óg kikke på korleis enkle matrisar og vektorar samhandlar.

Som du allereie har lært, lagar vi nye datatypar i C++ ved å deklarere klasser. Difor ynskjer vi å lage ei klasse for å inneholde og representere matrisar i programma våre.

Angåande Objektorientering: ObjektOrientert Programmering (OOP) er ein måte å strukturere programmane dine i delar eller klasser som vi kallar objekt. Dei kallast det sidan dei er tenkt å vere analoge med dei objekta dei representerar i den verkelege verda.

a) Lag ei klasse kalla `Matrix2x2`, som skal innehalde lagringsplass for 4 tal av typen double.

Når du vel korleis du skal lagre dataene dine er det ofte nyttig å vite korleis dei kjem til å verte nytta seinare. I matriseoperasjonar kjem vi til å aksessere dei forskjellege elementane av matrisa basert på rada og/eller kolonna deira. Difor er det nyttig å ordne medlemma i `Matrix2x2` som eit array.

b) Lag *set*- og *get*-funksjonar for matrisa.

```
double get(int row, int column) const;
void set(int row, int column, double value);
```

Angåande Objektorientering: I OOP har vi ein designkonvensjon angåande synlegheita til medlemmar. Synlegheit deklarerast med lagringsspesifikasjonane **public**, **private** (og seinare **protected**). Som ein tommelfingerregel skal alle medlemsvariablar vere private, og alle medlemsfunksjonar bør vere public. Sjølv om det finst bruksområder der det er praktisk å ha private medlemsfunksjonar og public medlemsvariablar.

Om du ikkje har gjort det allereie, forsikre deg om at medlemsvariablane dine er private og at get/set-funksjonane er public.

c) Test funksjonane dine.

Lag eit main-program, lag ei matrise, og skriv ut verdiane til matrisa.

Tips: Det er antagelegvis nyttig å lage ein print-funksjon for å vise elementane i matrisa på ein strukturert måte. Denne funksjonen kan vere anten ein medlemsfunksjon eller ein normal funksjon.

d) Lag ein standardkonstruktør. (default constructor)

Som du kanskje har lagt merke til i den forrige oppgåva, får du søppeldata ut om du prøver å skrive ut ei matrise der elementane ikkje har vorte eksplisitt tilordna. Måten å løse dette problemet på er å eksplisitt lage ein standardkonstruktør. Den bør sette elementa i matrisa til å vere lik *Identitetsmatrisa.* Identitetsmatrisa er ei matrise som har 1 langs øvre-venstre/nedre-høyre diagonalen, og 0 ellers.

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

e) Lag ein hjelpekonstruktør (convenience constructor)

Det vert fort eit irritasjonsmoment å måtte skrive fire linjar kode kvar gong du ynskjer å lage ei ny matrise. Som programmerar bør du styre unna ekstraarbeid av fleire årsakar... Difor, lag ein hjelpekonstruktør som tar alle fire elementa til matrisa som parametrar, og sett dei på riktig plass.

2 Operatoroverlasting (30%)

I denne oppgåva skal vi legge til nokre overlasta operatorar, slik at vi kan skrive enkle uttrykk med matrisane våre på ein kortare, og meir lesbar form, for eit litt meir inngåande eksempel på kvifor operatoroverlasting er nyttig, sjå vedlegget bakerst i øvinga.

Nokre av operatorane her vil trenge «Call-by-reference», sjå også vedlegget for ein kort gjennomgang av dette.

a) Overlast *operator* **<<, som skal skrive ut innhaldet i matrisa til outstream.** *Tips: Å overlaste denne operatoren tidlig vil gjere livet ditt enklare når du skal teste ei klasse du lagar.* **Drøft:**

- Kvifor bør denne funksjonen ta `const` argument? (t.d. `const Matrix2x2 & rhs`)
- Når bør vi, og når bør vi ikkje (evt kan ikkje) nytte `const`-argumentar?

b) Overlast operator+= og operator-=

Summen av to matriser er den elementvise summen.

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

Liknande når ein trekker to matrisar frå kvarandre, gjer vi ein elementvis subtraksjon.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} - \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} a-e & b-f \\ c-g & d-h \end{bmatrix}$$

Reflekter over følgjande:

- Kvifor kan vi *ikkje* berre skrive:

```
*this = *this + rhs;
```

- Kvifor kan du nytte += og -= operatorane når du arbeidar med *elementane* i matrisa?

c) Opprett følgjande matrisar i `main`:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} B = \begin{bmatrix} 4 & 3 \\ 2 & 1 \end{bmatrix} C = \begin{bmatrix} 1.0 & 3.0 \\ 1.5 & 2.0 \end{bmatrix}$$

Test følgjande uttrykk ved å nytte += og -=, skriv ut begge resultatane:

$$A+ = B$$

$$A- = B$$

Forsikre deg om at A har sin opprinnelege verdi etter testen.

d) Overlast +, -, og * operatorane. Produktet av to 2x2 matrisar kan reknast ut slik:

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

Tips:

- *Når du overlastar + og -, kan du nytte dei allereie overlasta operatorane += og -=. Å unngå duplisering av kode er som regel ein god ting.*
- *Produktet av to matriser kan skrivast som ei trippel for-løkke. (Noko som kan vere nyttig å kunne til Øving 6)*

e) Test implementasjonen din på matrisane vi lagde i c med følgjande uttrykk:

$$D = A + B$$

$$D = D * B - A + C$$

og skriv ut resultatet.

f) Overlast operator*=.
Tenk over følgjande:

- Kvifor kan du ikkje implementere *= på same måte som vi implementerte += og -= tidlegare?
- Kan du implementere *= ved å gjenbruke noko kode du har skrive allereie?

g) Skriv om testen fra e) til å nytte *= operatoren, slik at du også testar *=.

3 Vektor - Matrise-samhandling(15%)

Frå lineær algebra er det mange ting som er like ved matrisar og vektorar, eksempelvis kan ein behandle ein vektor som ei matrise med berre ei rad eller berre ei kolonne. Dette gjer det mogleg for oss å definere matrise-produktet mellom ei matrise og ein vektor som følgjar: (Vi bryr oss ikkje med radvektorar intill vidare)

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

a) Lag ei ny klasse `Vector2` som held ein to-dimensjonal matematisk vektor.
NB: Det er her snakk om ein matematisk vektor, og *ikkje* `std::vector` frå standardbiblioteket til C++.
Forsikre deg om at designet ditt følgjar OOP-konvensjonane angåande innkapsling (encapsulation), og skriv nødvendige/passande get- og set-funksjonane.

b) Implementer følgjande medlemsfunksjonar:

- **Returnerer dot-produktet mellom this og rightHandSide**
  ```
  double dot(const Vector2 &rightHandSide) const;
  ```
- **Returnerer den kvadrerte lengden av vektoren**
  ```
  double lengthSquared() const;
  ```
- **Returnerer lengden av vektoren. Kvadratrota av verdien kan reknast ut med funksjonen sqrt(double) som finst i <cmath>-headeren.**
  ```
  double length() const;
  ```

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

Drøft:

- Kva betyr `const` som står bak disse funksjonshovudane?
- Når bør ein nytte `const` på denne måten?

c) Implementer `operator<<` for `Vector2`.

d) (Valfri) **Implementer +=, -=, +, - og (unær)- operatorane for `Vector2` klassa.**

Medan desse operatorane er nødvendige for en funksjonelt komplett implementasjon av ein vektor, vil vi se seinare i kurset at disse operatorane kan verte overflødige ved nytting av arv.

4 Meir Matrisefunksjonalitet (25%)

For å kunne gjere noko nyttig med matrisane og vektorane våre treng vi å implementere matrise-vektor-produktet, i tillegg til nokon få nyttige funksjonar i Matrix2x2.

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

a) Lag medlemsfunksjonen: det(), som reknar ut determinanten til matrisa.
Prototype:

```
double det() const;
```

Determinanten til matrise A er gitt som følgjer:

$$det(A) = a * d - b * c$$

b) Lag medlemsfunksjonen: inverse()
Denne funksjonen skal returnere inversen av matrisa (om ein slik finst).
Prototype:

```
Matrix2x2 inverse() const;
```

Inversen av ei matrise er gitt som følgjer:

$$inv(A) = \frac{1}{det(A)} * \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

Tips: Legg merkje til at inversen av ei matrise kan finne på å ikkje eksistere, ***om determinanten er 0 må vi ha ein måte å flagge dette på****. Det er mange moglege måtar å løyse dette på:*

- *Vel ein ugyldig verdi som skal returnerast ved feilaktige matrisar. (Eit døme er null-matrisa).*
- *Endre prototypen for* ***inverse*** *slik at den tar resultatmatrisa som referanseparameter, og returnerer ein bool. Du kan då nytte den boolske verdien til å avgjere om matrisa vart invertert på riktig måte.*
- *Nytte* ***throw*** *og* ***exception****, som dekkjast seinare i kurset*

Korleis du vel å løyse dette, er opp til deg.

c) Implementer operator* mellom Matrix2x2 og Vector2. Implementer matrise-vektor-produktet som vist i Del 3, resultatet skal vere ein Vector2.
Tenk over følgjande:

- Kvifor gir det ingen meining å implementere *= operatoren i dette tilfellet?

5 Sy alt saman (10%)

Eit system av lineære likningar kan uttrykjast som ei matriselikning. Dette er måten systemar av likningar med hundrevis (eller til og med millionar) av ukjente løysast.

Gitt eit system av likingar med to ukjente (anta a, b, c, d, p og q som gitt):

$$I : ax + by = p$$

$$II : cx + dy = q$$

Den korresponderande likninga er:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} * \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} p \\ q \end{bmatrix}$$

Der målet er å avgjere kva x og y er.

Ein måte å løyse dette systemet er å multiplisere båe sidar av likninga med inversen av matrisa $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$, som gir:

$$\begin{bmatrix} x \\ y \end{bmatrix} = inverse \begin{bmatrix} a & b \\ c & d \end{bmatrix} * \begin{bmatrix} p \\ q \end{bmatrix}$$

Oppgåva di no er å kombinere dette til eit program som tar inn a, b, c, d, p og q frå brukaren og skriv ut dei resulterande x, y, eller ei feilmelding om ingen løysing eksisterar.

Vedlegg

Operatoroverlasting

Ein nyttig funksjon ved C++ er moglegheita til å overlaste operatorar. Overlasta operatorar er på mange måtar akkurat det same som funksjonar; det er måten dei kallast på som skil dei.

Døme (Eksempel): Bruk av funksjon mot bruk av operator.

```
class DummyExample {
   double value;
public:
   double get() const { return value; }
   DummyExample add ( const DummyExample &rightHandSide) const {
      DummyExample returnValue;

      returnValue.value = this->get() + rightHandSide.get();
      return returnValue;
   }

   DummyExample operator+( const DummyExample &rightHandSide) const {
      DummyExample returnValue;

      returnValue.value = this->get() + rightHandSide.get();
      return returnValue;
   }

};
```

I dømet over, legg merke til kva som er likt mellom funksjonen *sum* og operator+. Dei er faktisk nesten identiske. Den verkelege forskjellen kjem berre til syne når funksjonen vert kalla. Som illustrert i koden under:

```
double s1 = sum(l, r);     // Kallar funksjonen
double s2 = l + r;         // Kallar operatoren
```

For definisjoner av medlemsoperatorar/ikkje-medlems-operatorar/friend operatorar, sjå pensumboka eller forelesningsnotatane.

Call-by-reference

Tidlegare i øvingsopplegget har vi nytta peikarar, og sett at om ein sender ting som peikar til ein funksjon, så vil funksjonen kunne endre på det som vert sendt inn på ein slik måte at endringa óg skjer i den opprinnelege variabelen. Til dømes vil ein funksjon som skal bytte om på to tal, kunne utnytte dette:

```
void swap(int *first, int *second) {
    int temp = *first;
    *first = *second;
    *second = temp;
}
```

Dette ville faktisk ikkje vere mogleg med Call-by-Value. Peikarar er ellers nyttig, då vi kan endre KVA dei peikar på, i tillegg til at dei óg har ei tett knytting til arrays. (som vi skal sjå meir av i neste Øving). Men, det er ikkje alltid vi treng meir enn det vi nyttar i `swap` over; nemlig det å kunne endre variablar som høyrar heime utanfor funksjonen. Til dette formålet har vi ein litt meir lettvektsvariant av peikarar, nemlig «referansar», om vi skriv om eksempelet over til å nytte referansar, ser det slik ut:

```
void swap(int &first, int &second) {
    int temp = first;
    first = second;
    second = temp;
}
```

Legg merke til at vi her ikkje lenger derefererar nokon peikar, dette er fordi ei referanse ikkje kan endra KVA den refererar til, og dermed alltid vil gje verdien den refererar til. Vær og merksam på at &-teiknet som nyttast her, **ikkje** har lik meining som når vi arbeidar med peikarar (altså, til å finne adressa til noko). Her nyttast det som ein måte å seie «dette er ei referanse». Funksjonar som tek inn referansar, og endrar på disse, vil altså endre på dei variablane som vert sendt inn, noko som er viktig i f.eks. implementasjon av operatorane +=, ++, –, osv, sidan disse endrar på ein av dei variablane dei tar inn.