Norges teknisk–naturvitenskapelige universitet
Institutt for datateknikk og informasjonsvitenskap

# TDT4102 Prosedyre og Objektorientert programmering Vår 2014

**Øving 6**

## Frist: DD-MM-YYYY

## Mål for denne øvinga:

- Pekere
- Dynamiske arrays
- Dynamisk minnebruk
- Arv

## Generelle krav:

- Bruk de eksakte navn og spesifikasjoner 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 7.3, 9 & 10, Absolute C++ (Walter Savitch)
- It's Learning notater

## MERK:

- Denne øvinga er større enn de foregående, pass på at du setter av nok tid.
- Alle spørsmål merket med **FET**-skrift skal besvares.
- Når man implementerer klasser er det vanlig å lage seg én .h-fil og én .cpp-fil per klasse (f.eks. hvis klassen heter «Car», lager man Car.cpp og Car.h, og skriver all koden for klassen «Car» i disse filene.) Følg denne normen i oppgaver hvor det bes om å skrive klasser.

# 1 En generell Matriseklasse (15%)

I øving 5 ble du introdusert for en enkel 2x2 matriseklasse. I denne øvinga skal vi implementere en mer generell matriseklasse, nemlig en MxN-matrise, altså en matrise med M rader, og N kolonner.

Senere i øvinga skal du også implementere en N-dimensjonal Vektor ved bruk av arv.

**a) Lag en klasse kalt `Matrix`**, den skal inneholde lagringsplass for M*N tall av typen `double`.

Når vi skal bestemme hvordan vi skal lagre dataene i klassen er det lurt å vite hvordan vi skal bruke dataene siden. I matriseoperasjoner kommer vi til å referere til elementene basert på rad og kolonne. Derfor er det nyttig å ordne medlemmene i matrisen som en tabell. I tillegg kommer vi til å ha behov for å forandre størrelsen på dataene vi har lagret i matrisen, derfor bør tabellen være dynamisk allokert.

Tenk over følgende:

- Hvordan skal du lagre dimensjonene til matrisen?
- Senere vil vi få behov for å ha «ugyldige matriser», altså matriser som ikke er i en gyldig tilstand. Hvordan kan du implementere dette?

*Hint: Det er mulig å merke matrisen som ugyldig uten å måtte ha ekstra variabler. (NULL-peker)*

**b) Lag følgende konstruktører for matrisen:**

`Matrix()`

- Default-konstruktør, skal initialisere matrisen til den ugyldige tilstanden.

```
explicit Matrix(unsigned int nRows)
```

- Skal konstruere en gyldig nRows x nRows-matrise, initialisert som identitetsmatrisen. (`explicit`-nøkkelordet dekkes ikke av pensum, men bør brukes her).

```
Matrix(unsigned int nRows, unsigned int nColumns)
```

- Skal konstruere en gyldig nRows x nColumns-matrise, initialisert som 0-matrisen. (Alle elementer lik 0)

`~Matrix()`

- Destruktøren til Matrix, skal frigi/slette alt dynamisk allokert minne.

**c) Lag set- og get-funksjoner for matrisen**

Get funksjonen skal hente ut verdien til ett element i matrisen, uttrykt ved rad og kollone. Tilsvarende skal set funksjonen sette verdien til ett element. *Det er også fordelaktig å lage get-funksjoner for høyden og bredden på matrisen*

**d) Lag medlemfunksjonen:** `isValid() const`

Denne funksjonen skal returnere `true` om matrisen er gyldig, og ellers `false`.

**e) Overlast** *operator* <<

For å gjøre testing enklere. Forsikre deg om at du har tatt høyde for ugyldige matriser.

**f) Test funksjonene dine.**

Lag et main-program, lag en matrise med hver av de tre konstruktørene, og skriv ut verdiene fra hver av de tre matrisene.

Test også at medlemsfunksjonene dine gjør det de skal.

# 2 Kopiering av objekter (15%)

Når vi har dynamisk allokert minne, må vi ordne enkelte ting selv, som før ble gjort automatisk for oss, i tillegg til å passe på allokering og deallokering av minne, må vi også bestemme hvordan objektene våre skal kopieres. Ta nå en kikk på vedlegget bakerst i oppgavesettet, for å se en forklaring av hvorfor og hvordan disse to oppgavene skal implementeres.

**a) Implementer `operator=` til Matrix-klassen, slik at du følger deep-copy-konseptet beskrevet i Vedlegget.**

**b) Implementer kopikonstruktøren til Matrix-klassen, slik at du følger deep-copy-konseptet beskrevet i Vedlegget.**

*Vær obs på at de påfølgende oppgavene ikke kan gjøres uten problemer hvis denne oppgaven ikke er gjort.*

# 3 Operatoroverlasting (10%)

**a) Overlast `operator+=` og `operator-=`**

*Hint: Legg merke til at for at resulatet av matriseoperasjonene skal være gyldig, må dimensjonene på matrisen på venstre side, være like dimensjonene til matrisen på høyre side.*

**b) Overlast +, - og * operatorene.**

Tips: Prøv å gjenbruke implementasjonene av += og -=.

*Hint: Produktet av to matriser kan skrives som trippel for-løkke.*

**c) Overlast `operator*=`.**

Tenk over følgende:

- Hvorfor kan du ikke implementere *= på samme måte som vi implementerte += og -= tidligere?
- Kan du implementere *= og forsikre deg om at dataene er konsistente, ved å gjenbruke noe kode du allerede har skrevet? Hvorfor stemmer dette?

**d) Test implementasjonen din.**

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

$$D = A + B$$

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

Fasit:

$$D = A + B = \begin{bmatrix} 5 & 5 \\ 5 & 5 \end{bmatrix} D = D * B - A + C = \begin{bmatrix} 30.0 & 21.0 \\ 28.5 & 18.0 \end{bmatrix}$$

# 4 Vektor - Matrise-samhandling (15%)

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 MxN-matriseklassen.**

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

NB: Det er her snakk om en matematisk vektor, og akkurat som i forrige øving **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);
```

- Kopi-konstruktør fra 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. Lag deretter en Matrix & som refererer til `this`, gjennom denne referansen har du tilgang til Matrix sin operator=.

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 initializer-list for å kalle Matrix sin konstruktør.*

**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::get(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`.

```
double lenghtSquared() 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().*

# 5 Forstå det du har gjort. (15%)

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! Hvordan? Vel, DU gjorde det!

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 enkelt-kolonnes vektor. I tillegg har du i denne øvinga latt Vector arve alle egenskapene ved Matrix, inklusive operatorene for * og *=. Dermed vil vektoren bli behandles om 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. 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.

# 6 Minesveiper (30%)

I denne deloppgaven skal du lage spillet Minesveiper. Minesveiper består av et rektangulært spillebrett som er ett rutenett av firkanter. Under et satt antall tilfeldige ruter ligger det miner og spilleren må åpne alle rutene som ikke inneholder miner for å vinne spillet. Hvis spilleren åpner en rute som inneholder en mine, har han tapt. Ruter som ikke inneholder en mine, har et nummer fra null til åtte som representerer hvor mange miner det finnes under naborutene. Siden spilleren skal ha mulighet for å stille vanskelighetsgraden til spillet selv må tabellene (arrays) som representerer spillebrettet være dynamiske, slik at spilleren kan bestemme størrelsen på brettet.

**a) Lag klassen `Minesweeper` som har følgende *private* medlemsvariabler:**

- `height`, et heltall som representerer høyden (antall rader) på brettet.
- `width`, et heltall som representerer bredden (antall koloner) på brettet.
- `mines`, et heltall som representerer antall miner på brettet.
- `open`, et heltall som representerer antall ruter spilleren har åpnet.
- `board`, en 2 dimensjonal dynamisk tabell av heltall som representerer spillebrettet
- `mask`, en 2 dimensjonal dynamisk tabell av heltall som representerer hvilke ruter spilleren har åpnet.

Du skal nå lage en del funksjoner som trengs i spillet. Bestem selv for hver funksjon om den skal være *private* eller *public*:

**b) Lag funksjonen `getInput` som ikke tar noen argumenter og ikke returnerer noe.**

Denne funksjonen lar spilleren skrive inn størrelsen på spillebrettet og antall miner og lagrer dette i *height*-, *width*-, og *mines*-variablene. Funksjonen må sjekke at tallene spilleren gir er gyldige, og spør etter nye tall hvis ikke. For at tallene skal være gyldige må høyde og bredde være minst 2, og antall miner være minst 1, men ikke større enn halvparten av rutene på brettet (høyde x bredde).

**c) Lag funksjonen `makeBoard` som ikke tar noen argumenter og ikke returnerer noe.**

Denne funksjonen skal lage de 2 dimensjonale tabellene (*arrays*) *board* og *mask* og fylle dem med null. Størrelsen på tabellene er gitt av *height* og *width*.

**d) Lag funksjonen `placeMines` som ikke tar noen argumenter og ikke returnerer noe.**

Denne funksjonen skal plassere et antall miner, gitt av medlemsvariabelen *mines*, i tabellen *board*. En mine kan representeres med verdien -1. Denne funksjonen skal også inkrementere alle naborutene **som ikke er miner** med én. Pass på at du ikke prøver å inkrementere et felt utenfor tabellen.

**e) Lag funksjonen `setFlag` som tar to heltall som argument og ikke returnerer noe.**

Denne funksjonen lar spilleren markere en rute, representert ved de to gitte heltallene, som han tror inneholder en mine. En markert rute, eller et flagg, kan representeres ved å sette det tilsvarende feltet i mask til 2. Hvis en rute allerede er markert, skal funksjonen fjerne markeringen (sette feltet i *mask* til 0). Hvis ruten er åpnet (feltet i *mask* er 1), eller hvis de gitte argumentene representerer en rute utenfor tabellen, skal funksjonen ikke gjøre noe.

**f) Lag funksjonen `openSquare` som tar to heltall som argument og returnerer en *bool*.**

Denne funksjonen skal åpne en rute, gitt av argumentene, på brettet. Det vil si, den skal sette det tilsvarende feltet i mask til 1. Ruten skal kun åpnes hvis de gitte argumentene ikke representerer en rute utenfor tabellen og ruten er uåpnet og ikke markert. Hvis ruten inneholder en mine, skal funksjonen returnere **true** for å indikere at spilleren har tapt. I alle andre tilfeller skal funksjonen returnere **false**. Hvis ruten ble åpnet, og ikke inneholdt en mine, skal medlemsvariabelen open inkrementeres.

**g) Lag funksjonen `printBoard` som tar en *bool* som argument og ikke returnerer noe.**

Denne funksjonen skal skrive ut spillebrettet til skjermen på en oversiktelig måte. Finne en måte å representere:

- uåpnede ruter
- åpne ruter, disse skal skrives som tallet som representerer antall nabominer. Hvis tallet er null kan du velge å skrive det som en tom rute for at brettet skal se ryddigere ut.
- markerte ruter
- miner, disse skal kun vises hvis argumentet er **true**

**h) Lag funksjonen `cleanUpMemory` som ikke tar noen argumenter og ikke returnerer noe.**

Denne funksjonen skal deallokere minnet som brukes av de dynamiske tabellene.

*Hint: Siden en 2 dimensjonal dynamisk tabell kan betraktes som en tabell av tabeller (array of arrays), må du først bruke delete[] på de innerste tabellene, før du bruker den på den ytterste.*

**i) Lag funksjonen `play` som ikke tar noen argumenter og ikke returnerer noe.**

I denne funksjonen skal du sette sammen spillet. Bruk funksjonene du har laget, sammen med litt ekstra kode for å lage ferdig et spill som lar spilleren bestemme størrelsen på spillebrettet og antall miner, for deretter å sette og fjerne markeringer og åpne ruter inntil han enten har åpnet alle rutene uten miner (vunnet), eller åpnet en rute med en mine (tapt). Når spilleren har vunnet eller tapt, skal spillet spørre spilleren om han vil spille igjen og, hvis han ønsker å spille igjen, spør etter ny størrelse på brett og antall miner.

**j)** *(valgfri)* NB! Denne deloppgaven bruker en teknikk som det kanskje ikke er forelest om ennå: Rekursjon.

Hvis du har spilt Minesveiper før (som du sikkert har) har du kanskje lagt merke til at hvis du åpner en rute som ikke er en nabo til en mine (det vil si, feltet i board-tabellen er null), vil alle naboene til den ruten også bli åpnet, noe som gjør at du kan åpne store deler av brettet på en gang.

For å implementere denne funksjonaliteten i spillet ditt må du gjøre følgende: **Endre funksjonen *openSquare* slik at hvis spilleren åpner en rute med verdi 0 vil alle naboene til den ruten også åpnes.** For å implementere dette på en enkel måte, kan du bruke rekursjon. La *openSquare* kalle seg selv for hver nabo av ruten spilleren åpner, hvis ruten ikke har en mine som nabo.

## Vedlegg: Forklaring av kopikonstruktor og operator=

Det oppstår et spesialtilfelle vi må ta hensyn til når vi har dynamisk allokerte medlemsvariabler i en klasse.

Se på følgende eksempel:

```
class Example {
   private:
     int *anInt;

   public:
     Example(int i) : anInt(0) {
        anInt = new int(i);
     }

     Example() : anInt(0) {
        anInt = new int(0);
     }

     ~Example() {
         if (anInt) {
            delete anInt;
         }
     }

     int get() const { return *anInt; }
};
```

Ved første øyekast ser dette ut som en veldefinert klasse. Den konstrueres riktig, og dersom konstruktøren fullfører kan vi være sikre på at vi alltid har et initialisert heltall lagret i `anInt`. Problemet vi ser etter dukker først opp når vi skriver denne koden:

```
Example a(5);
Example b;

b = a; // Hva skjer her?
```

Hva skjer i koden over når vi skriver `b = a`? Jo, et operator-kall, til operator=, men vi har da ikke definert noen slik operator. Operatoren som blir kalt er en som kompilatoren lager automatisk, denne operatoren tar en binær kopi. Dvs. den kopierer medlem for medlem uten å bry seg om noe er dynamisk allokert.

Hvorfor er dette et problem? Det er et problem fordi den eneste medlemsvariabelen i klassen vår, Example, er en peker. Dermed er det *pekeren* og IKKE minnet pekeren peker til som blir kopiert. Resultatet er at `b` og `a` har medlemmer som peker til nøyaktig samme instans av det dynamisk allokerte heltallet!

Så, la oss ta en kikk på følgende kodesnutt:

```
Example a(5);
if (a.get() > 0) {
    Example b = a;
    cout << b.get() << endl;
}
```

Hva skjer i koden over?

Først legger vi merke til at vi kommer til å gå inn i if-setningen, siden vi vet at a er 5. Så kopieres `a` til `b`, `b` skrives ut, og i det vi forlater blokken rundt if-setningen vil `b` bli destruert. Dvs, `b` sin destruktør vil bli kalt. Husk fra definisjonen av klassen at destruktøren sletter minnet som er dynamisk allokert og pekt på av `anInt`.

Hva skjedde med a sin anInt? Siden b og a hadde pekere til det samme dynamisk allokerte heltallet, vil pekeren i a nå være ugyldig.

Hvordan kan vi fikse dette? Det som vi har beskrevet over kalles en «grunn kopi» («shallow copy»), og er ikke alltid en dårlig løsning. Men, man må isåfall holde nøye orden på hvor mange objekter som deler samme minne o.l. og det er utenfor pensum for dette kurset.

Løsningen er å implementere en algoritme kalt «dyp kopi» («deep copy»). En dyp kopi vil allokere det minnet som trengs for eventuelle dynamisk allokerte medlemsvariabler dynamisk når man kaller på operator=, for så å eksplisitt initialisere de nylig allokerte variablene.

For å implementere «deep copy» for Example-klassen vår, må vi legge til en eksplisitt implementasjon av `operator=` og *kopi-konstruktøren* (copy constructor):

```
Example &operator=(const Example &rhs) {
    // Siden alle konstruktoerer garanterer at vi har allokert minnet vi trenger
    // for aa lagre *anInt, kan vi helt enkelt bare tilordne den.
   *(this->anInt) = *(rhs.anInt);
}
// Dette er kopikonstruktoeren, den har alltid formen:
// classname(const classname&).
// Det er denne konstruktoeren som kalles naar du initialiserer
// et objekt med et annet objekt av samme type.
// Som for eksempel:
// Example a(5);
// Example b(a); //Dette kaller kopikonstruktoeren til b.

Example(const Example &other) : anInt(0) {
   this->anInt = new int();
   *(this->anInt) = *(other.anInt);
}
```