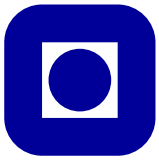

Norges teknisk–naturvitenskapelige universitet
Institutt for datateknikk og informasjonsvitenskap

TDT4102 Prosedyre og Objektorientert programmering Vår 2015

Øving 6

Frist: 2015-02-27

Mål for denne øvinga:

- Pekere
- Dynamisk minnebruk
- Dynamiske arrays

Generelle krav:

- Følg spesifikasjonene og bruk klasse- og funksjonsnavn 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 7.3, 9 & 10, Absolute C++ (Walter Savitch)
- It's Learning notater

MERK:

- 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.) Du bør følge denne normen i oppgaver der det bes om å skrive klasser.

1 Dynamisk minnehåndtering (20%)

I denne deloppgaven skal vi lage noen funksjoner for å bli kjent med dynamisk minnehåndtering. For en kjapp oppfriskning av hvorfor dette er nyttig, og hvordan dette gjøres, kan du se i vedlegget bakerst i øvinga.

a) **Lag en funksjon `void fillInFibonacciNumbers(int result[], int length)`** Funksjonen skal regne ut `length` tall i Fibonacci-tallrekka, og lagre disse i tabellen `result`. (NB! Denne funksjone skal bare ta inn en tabell og fylle denne med tall.)

Om fibonacci tallrekken:

- Fibonacciserien er serien av tall som starter med 0 og 1 og hvor hvert påfølgende element er summen av de to foregående: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 osv.

b) **Lag en funksjon `void printArray(int arr[], int length)`** Funksjonen skal skrive ut `length` elementer fra tabellen `arr` (starter på indeks 0).

c) **Lag en funksjon `void createFibonacci().`** Funksjonen skal be brukeren om lengden på tallrekka, dynamisk allokere en tabell av riktig størrelse, fylle denne med tall og skrive ut. Funksjonen kan oppføre seg omtrent som dette, erstatt kommentarene med din kode:

```
void createFibonacci() {
    // 1. Spør brukeren hvor mange tall som skal genereres
    // 2. Alloker minne til en tabell som er stor nok til tallrekka
    // 3. Fylle tabellen med funksjonen fillInFibonacciNumbers
    // 4. Skriv ut resultatet til skjerm med printArray
    // 5. Frigjør minnet du har reservert
}
```

d) **Test funksjonene ved å kalle createFibonacci fra main()**

2 En generell Matriseklasse (25%)

I øving 5 ble du introdusert for en enkel 2x2 matriseklasse. I denne øvinga skal vi implementere en mer generell matriseklasse som gjør det mulig å ha matriser av forskjellig størrelse – en matrise med M rader og N kolonner hvor M og N spesifiseres når objektene instansiseres.

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

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 matriser av forskjellig størrelse, derfor bør tabellen dynamisk allokereres når objektene instansieres.

Tenk over følgende:

- Det er to forskjellige teknikker for å lage dynamisk allokerte todimensjonale tabeller:
  - Du kan velge mellom å allokere en endimensjonal tabell og regne om fra todimensjonal indeks til endimensjonal indeks.
  - Du kan brukere peker-til-peker teknikken.
- Det er også behov for å kunne si at matriser er «ugyldige», altså matriser som ikke er i en gyldig tilstand. En matrise som ikke er initialisert er en ugyldig matrise, og resultatet fra en aritmetisk operasjon på to matriser av forskjellig størrelse skal også returnere en ugyldig matrise.
  Hint: Det er mulig å merke matrisen som ugyldig med bruk av ***`nullptr`****-peker. Da trenger du ikke en ekstra variabel.*

b) Lag følgende konstruktører for matrisen:

`Matrix()`

- Default-konstruktør som skal initialisere matrisen til den ugyldige tilstanden. I praksis skal du ikke allokere minne i denne konstruktøren. Det eneste du må passe på er at tabell-pekeren settes til null. I C++ v11 bruker vi `nullptr`, mens i tidligere versjoner måtte du bruke 0 eller NULL.

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

- Skal konstruere en gyldig nRows x nColumns-matrise, initialisert som 0-matrisen (alle elementer er lik 0). Her må du selvsagt allokere en matrise og initialisere verdiene.

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

- Skal konstruere en gyldig nRows x nRows-matrise, initialisert som identitetsmatrisen. I lineær algebra er identitetsmatrisen, eller enhetsmatrisen, en n x n matrise med verdien 1 på hoveddiagonalen og 0 på de resterende plassene (se:`http://no.wikipedia.org/wiki/Identitetsmatrise`. Identitetsmatrisen har samme funksjon i matrisemultiplikasjon som tallet 1 i vanlig multiplikasjon. (`explicit`-nøkkelordet dekkes ikke av pensum, men bør brukes her). *Hint: Du kan bruke en delegerende konstruktør for å forenkle denne deloppgaven (C++11). Dvs. at du gjør at kall til konstruktøren som du implementerte over.*

```
~Matrix()
```

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

c) Lag set- og get-funksjoner for matrisen

```
double get(unsigned int row, unsigned int col) const
void set(unsigned int row, unsigned int col, double value)
```

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.

3 Et praktisk eksperiment med kopier og tilordning(20%)

Merk at vi i denne oppgava skal fremprovosere noen feil, så programmet ditt kan, og skal oppføre seg litt rart.

a) Lag deg to matriser A og B i main med forskjellige verdier, forsøk å tilordne A til B ved å skrive `A = B;`
Teorispørsmål: Hvilken operator brukes her?
Teorispørsmål2: Hva skjer med verdiane i A, om du no endrar på verdiane i B?

Nå vil programmet ditt høyst sannsynlig kræsje på slutten av main, dette er forventet, ettersom A og B sine pekere peker på samme tabell, dette skal vi løse i oppgave 4.

b) Lag deg en tredje matrise C, ved å skrive `Matrix C(A);`
Teorispørsmål: Hvilken funksjon brukes her?

c) Lag deg en fjerde matrise D, ved å skrive `Matrix D = B;`
Teorispørsmål: Hvilken funksjon brukes her?

For å se hva som foregår i koden du har skrevet, kan vi først fjerne default-implementasjonene til tilordningsoperatoren. Dette kan du gjøre ved å legge til deklarasjonen for operator= i klassen, men skrive `= delete` etter deklarasjonen. Da forteller vi kompilatoren at den IKKE skal opprette default implementasjon av denne operatoren.

d) Legg til følgende deklarasjon i klassen: `Matrix &operator=(Matrix rhs) = delete;`.
Se på hvilke linjer i koden din som nå får feil, disse linjene er de som bruker operator=

e) Gjenta operasjonen over for kopikonstruktøren, ved å legge til denne i klassedefinisjonen med `=delete`
For å spare deg for forvirring, kan du kommentere ut linja du la til i forrige oppgave, slik at du bare ser hvor kopikonstruktøren brukes.

f) Teorispørsmål: Stemte det du nå ser med antagelsene dine fra oppgavene a, b og c?

g) Merk forøvrig også at samtlige steder som tar inn Matrix by-value, eller returnerer Matrix by value bruker kopi-konstruktøren. **Prøv ut dette ved å lage en funksjon som har Matrix-typen som parameter og Matrix som retur-type**.

Du kan nå fjerne `=delete` fra både operator= og kopi-konstruktøren, ettersom du i neste oppgave skal lage egne implementasjoner av disse.

4 Overlaste kopiering og tilordning (20%)

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 dde følgende oppgavene skal implementeres.

a) Implementer kopikonstruktøren til Matrix-klassen, slik at du følger deep-copy-konseptet beskrevet i vedlegget.

```
Matrix(const Matrix & rhs)
```

Kopikonstruktøren skal sørge for at det blir instansiert en ny matrise av samme størrelse som rhs og at alle verdier blir kopiert fra rhs.

b) Implementer `operator=` til Matrix-klassen, slik at du følger deep-copy-konseptet beskrevet i vedlegget. Operatoren din må ta hensyn til at matrisen på begge sider kan være ugyldig eller kan være av andre dimensjoner enn den andre matrisen. Pass på at operatoren din ikke lekker minne. Den enkleste måten å implementere dette på er å bruke copy-swap paradigme som er forklart i vedlegget.

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

5 Operatoroverlasting (25%)

a) Overlast `operator+=` Dersom matrisene er av ulike dimensjoner, skal lhs settes til den ugyldige matrisen.

b) Overlast `operator+` Dersom brukeren forsøker å gjøre en ulovlig operasjon, f.eks. ved å summere matriser av forskjellige dimensjoner, skal operatorene returnere en *ugyldig* matrise.
Tips: Prøv å gjenbruke implementasjonene av +=. Bruk også gjerne kopikonstruktøren.

c) Test løsningen din

$$A = \begin{bmatrix} 1.0 & 2.0 \\ 3.0 & 4.0 \end{bmatrix} B = \begin{bmatrix} 4.0 & 3.0 \\ 2.0 & 1.0 \end{bmatrix} C = \begin{bmatrix} 1.0 & 3.0 \\ 1.5 & 2.0 \end{bmatrix}$$

$$A+ = B + C$$

Fasit:

$$A+ = B + C = \begin{bmatrix} 6.0 & 8.0 \\ 6.5 & 7.0 \end{bmatrix}$$

d) (Valgfritt) Implementer de andre operatorene fra øving 5 (-, -=, * og *=)

Vedlegg: Forklaring av dynamisk minne, kopikonstruktor og operator=

Dynamisk minnehåndtering

Hittil i øvingsopplegget har størrelsen på alle variabler og tabeller vært kjent idet programmet kompileres. Dermed kan kompilatoren generere maskinkode som reserverer korrekt mengde minne til alle deler av programmet, og også frigjøre minnet når variablene går ut av scope.

Ofte vil vi ønske å ha håndtere en mengde data vi ikke vet før programmet kjører. For eksempel kan vi ønske å bruke forskjellig mengde minne avhengig av input fra bruker. At vi bestemmer hvor mye minne som skal reserveres mens programmet kjøres er årsaken til at dette kalles «dynamisk», og står i motsetning til «statisk» minnehåndtering.

Vi kan reservere minne mens programmet kjører, men vi kan ikke lage nye variabelnavn «på sparket». Derfor er pekere vesentlig når vi skal bruke dynamisk minne. For å reservere dynamisk minne i C++ bruker vi operatoren `new`, som returnerer en peker til begynnelsen av det minnet programmet har fått tildelt. Uten å ta vare på denne pekeren i en pekervariabel har vi ingen måte å lese fra eller skrive til dette minnet.

Merk! Når vi reserverer minne med `new` kan ikke kompilatoren vite hvor lenge programmet har bruk for minnet, og vi må selv ta ansvar for å rydde opp etter oss, og det gjør vi med `delete`-operatoren:

```
void newAndDelete() {
    // Allocate memory for an int using new
    int *x = new int {};

    cout << "Please input an integer: ";
    cin >> *x;
    cout << "Well done! You wrote: " << *x << endl;

    // Release the memory when we're done with it
    delete x;
    x = nullptr;
}
```

Dersom man ikke gjør dette vil det over programmets levetid bli allokert mer og mer minne som ikke blir frigitt, helt til datamaskinen er tom for fysisk minne og programmet kræsjer. Dette kalles en minnelekkasje. Det er ganske lett at det oppstår minnelekkasjer når man bruker dynamisk minne, så beste praksis er å kun bruke det der det er absolutt nødvendig. (Eksempelet over er bruk av dynamisk minne helt unødvendig, men det er kun ment som et trivielt eksempel for å demonstrere syntaksen.)

Der man må bruke det er konvensjonen at den delen av programmet som reserverer minnet «eier» minnet og er derfor også ansvarlig for å frigjøre minnet. I prosedural kode vil det som regel bety samme funksjon, og i objektorientert kode betyr det at man allokerer minne i konstruktøren til et objekt og frigjør det i destruktøren.

Dersom man skal allokere minne til et `array` gjøres dette slik:

```
void dynamicArrayExample() {
    int n = 0;
    cout << "How many numbers do you want to type? ";
    cin >> n;

    // Allocate memory for double array of size n
    double *numbers = new double[n] {};
    for(int i = 0; i < n; i++) {
        cout << "Input number: ";
        cin >> numbers[i];
    }

    cout << "You entered: ";
    for(int i = 0; i < n; i++) {
        cout << numbers[i] << " ";
    }
    cout << endl;

    // Release the memory when we're done with it
    delete[] numbers;
    numbers = nullptr;
}
```

NB! Legg merke til at når vi frigjør minnet som er reservert for et `array` så må vi bruke `delete[]` og ikke `delete`!

Introdukjon til klasser med dynamisk allokerte medlemsvariabler

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 = new int(i);
     }

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

     ~Example() {
```

```
        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 begge har en peker som peker til nøyaktig samme instans av det dynamisk allokerte heltallet!

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.

Kopikonstruktør og operator=

For å implementere «deep copy» for Example-klassen vår, må vi legge til en eksplisitt implementasjon av *kopikonstruktøren* og `operator=`.

Kopikonstruktøren har alltid formen `Klassenavn(const Klassenavn &)`. Det er denne som kalles når du initialiserer et objekt med et annet objekt av samme type, som for eksemepl: `Example b(a);`

```
Example(const Example &other) : anInt(nullptr) {
   this->anInt = new int(); //vi allokerer det vi trenger av minne
   *anInt = other.get();   //vi kopierer verdier fra other til this
}
```

Tilordningsoperatoren kan imlementeres på forskjellige måter, men en vanlig (og anbefalt) teknikk å bruke er "copy-and-swap". Kort fortalt bruker vi en call-by-value parameter for høyreoperanden, da tar vi inn en kopi (`rhs`) som er opprettet med kopi-konstruktøren.

Deretter swapper vi alle medlemsvariabler (inkl. pekere til allokert data) mellom de to objektene. Dermed får `this` (venstresiden) tilordnet høyresidens verdier (og allokerte data) og `rhs` ender opp med det som skal slettes. Siden `rhs` er en lokal variabel vil destruktøren dens automatisk bli kalt når den går ut av scope.

```
Example &operator=(Example rhs) {
   //vi tar inn rhs som call-by-value
   std::swap(anInt, rhs.anInt) //vi swapper pekere
   return *this;
}
```