



**Frist: 2016-03-04**

### Mål for denne øvingen:

- Dynamisk minnehåndtering med pekere og smartpekere
- Implementering av kopikonstruktører
- Mer om operatoroverlasting

### 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, compilere og kjøre.

### Anbefalt lesestoff:

- Kapittel 7.3, 9 & 10, Absolute C++ (Walter Savitch)

*NB: 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 (15%)

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 øvingen.

- a) **Lag en funksjon `void fillInFibonacciNumbers(int result[], int length)`.** Funksjonen skal regne ut `length` antall tall i Fibonacci-tallfølgen og lagre disse i tabellen `result`. Fibonaccifølgen er følgen av tall som starter med 0 og 1 og der hvert påfølgende element er summen av de to foregående: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 osv. Funksjonen skal ikke selv gjøre noe minnehåndtering, og skal lagre verdiene i tabellen som tas inn som argument.
- 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. Allokere minne til en tabell som er stor nok til tallrekka  
    // 3. Fyll tabellen med funksjonen fillInFibonacciNumbers  
    // 4. Skriv ut resultatet til skjerm med printArray  
    // 5. Frigjør minnet du har reservert  
}
```

Test funksjonene ved å kalle `createFibonacci` fra `main()`.

## 2 En generell matriseklasse (25%)

I øving 6 lagde vi en enkel klasse for å representere  $2 \times 2$ -matriser. I denne øvingen skal vi implementere en mer generell matriseklasse for å gjøre det mulig å representere matriser av vilkårlig størrelse.

- a) **Lag en klasse kalt `Matrix`.** Klassen skal inneholde lagringsplass for  $M \times N$  tall av typen `double`, der  $M$  og  $N$  er to vilkårlige heltall større enn 0.

I matriseoperasjoner kommer vi til å henvise til elementene i matrisen etter rad og kolonne. Derfor er det nyttig å ordne medlemmene i matrisen som en tabell. Siden klassen skal kunne representere en tabell av vilkårlig størrelse, må tabellen dynamisk allokteres ved instansiering.

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 en matrise er «ugyldig», altså at matrisen 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 ved å bruke en `nullptr`-peker. Der-  
som du gjør dette trenger du ikke en ekstra variabel for å markere matrisen som ugyldig.*

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

`Matrix()`

- En *standardkonstruktør* (default constructor) 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 tabellpekeren settes til en ugyldig verdi. I C++11 (som det er støtte for i både VS 2015 og nyere Xcode) gjør vi dette ved å sette pekeren lik `nullptr`, mens vi i tidligere versjoner ville brukt 0 eller `NULL`.

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

- Skal konstruere en gyldig  $nRows \times nColumns$ -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)`

- Denne skal konstruere en gyldig  $nRows \times nRows$ -matrise, initialisert som identitetsmatrisen. I lineær algebra er *identitetsmatrisen* (også kjent som enhetsmatrisen) en  $N \times N$  matrise med verdien 1 på hoveddiagonalen og 0 på de resterende plassene. Identitetsmatrisen har samme funksjon i matrisemultiplikasjon som tallet 1 i vanlig multiplikasjon: når en kvadratisk matrise multipliseres med enhetsmatrisen får man den opprinnelige matrisen som svar.

Vi bruker her `explicit`-nøkkelordet, som ikke dekkes av pensum. Dette nøkkelordet gjør at den aktuelle konstruktøren ikke kan brukes til å «automatisk» konvertere andre typer til den aktuelle klassen. I denne oppgaven bruker vi `explicit` for å unngå potensielle feil som kan være vanskelig å oppdage.

`~Matrix()`

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

For å forenkle denne oppgaven kan du gjenbruke kode for å lage de ulike konstruktørene. Imidlertid er det generelt smart å ikke «copy-paste» kode rundt omkring i kodebasen din. For å unngå dette kan man for eksempel bruke en felles medlemsfunksjon som begge konstruktørene kaller. I C++11 kan man også bruke en *delegerende konstruktør*. Her er et eksempel bruk av en slik konstruktør:

```

class Eksempel {
    public:
        Eksempel(int a) {
            // Kode for konstruktør én
        }

        Eksempel(int a, int b) : Eksempel(a) {
            // Kode for konstruktør to
            // Konstruktør én brukes for å behandle
            // det første argumentet
        }
}

```

- 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. Funksjonene skal ha følgende prototyper:

```

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

```

- d) **Lag medlemsfunksjonene getHeight og getWidth.** Disse skal henholdsvis returnere matrisens høyde og bredde.
- e) **Lag medlemsfunksjonen isValid() const.** Denne funksjonen skal returnere true om matrisen er gyldig, og ellers false.
- f) **Overlast operator <<.** Å overlaste denne operatoren lar oss bruke Matrix med cout, som gjør testing av koden vår enklere. Pass på at du håndterer ugyldige matriser.
- g) **Test funksjonene dine.** Lag en main-funksjon, og opprett i den matriser med hver av de tre konstruktørene. Skriv så ut verdiene fra hver av de tre matrisene. Test også at medlemsfunksjonene til Matrix-klassen gjør det de skal.

### 3 Et praktisk eksperiment: kopier og tilordning (20%)

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

- a) **Opprett to matriser A og B i main med forskjellige verdier. Forsøk så å tilordne A til B ved å skrive A = B. Svar så på følgende spørsmål:**

- Hvilken operator brukes her?
- Hva skjer med verdiene i A dersom du nå endrer på verdiene 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. I Oppgave 4 skal vi løse dette.

- 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 standardimplementasjonene 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 standardimplementasjonen av denne operatoren.

- d) Legg til følgende deklarasjon i klassen: `Matrix &operator =(Matrix rhs) = delete;`.  
Se på hvilke linjer i koden din som nå gir feil. Disse linjene er de som bruker `operator =`.
- e) Gjenta operasjonen over for kopikonstruktøren, ved å legge til denne i klasse-definisjonen med `= delete`.  
For å spare deg for forvirring kan du kommentere ut linjen 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 kopikonstruktøren brukes overalt i koden der du tar inn `Matrix` «by value» eller returnerer `Matrix` «by value». Sjekk dette ved å lage en vilkårlig funksjon som tar inn en parameter av typen `Matrix` og har `Matrix` som returtype. Hva skjer, og hvorfor?

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. Vedlegget bakerst i øvingen gir en mer utfyllende beskrivelse av hvorfor og hvordan de følgende oppgavene skal implementeres.

- a) **Implementer kopikonstruktøren til Matrix-klassen.** Konstruktøren skal implementeres i henhold til prinsippene bak dyp kopiering (deep copying), som 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`. Hvis `rhs` er en ugyldig matrise skal selvsagt kopien også være en ugyldig matrise.

- b) **Implementer operator = for Matrix-klassen.** Operatoren skal implementeres i henhold til prinsippene bak dyp kopiering (deep copying), som beskrevet i vedlegget.

Pass på at operatoren din ikke lekker minne. Den enkleste måten å implementere operatoren riktig på er å bruke copy-and-swap-teknikken, som er forklart i vedlegget.

*NB: De påfølgende oppgavene kan ikke gjøres uten problemer hvis denne oppgaven ikke er gjort.*

## 5 Operatoroverlasting (20%)

- a) **Overlast operator +=.** Dersom matrisene er av ulike dimensjoner, skal resultatet av operator-kallet bli en ugyldig matrise. Dvs. venstre operand skal bli en ugyldig matrise og returverdien skal være en ugyldig matrise.
- b) **Overlast operator +.** Dersom brukeren forsøker å gjøre en ulovlig operasjon, f.eks. ved å summere matriser av forskjellige dimensjoner, skal operatoren returnere en ugyldig matrise. Husk at du tidligere i øvingen skal ha lagt til mulighet for å markere en matrise som ugyldig.  
*Tips: Prøv å gjenbruke implementasjonen av +=. Bruk også gjerne kopikonstruktøren.*
- c) **Test løsningen din.** Definer matrisene `A`, `B` og `C` som følger, og sjekk at du får svar som forventet av `A += B + C`.

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

- d) (Valgfritt) **Implementer operatorene -, -=, \* og \*=-.**

## Vedlegg: Forklaring av dynamisk minne, kopikonstruktør 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 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», 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 at 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. Det gjør vi med `delete`-operatoren:

```
void newAndDelete() {
    // Allokert minne for en int med new
    int *x = new int {};

    cout << "Skriv inn et heltall: ";
    cin >> *x;
    cout << "Takk! Du skrev: " << *x << endl;

    // Frigjør minnet når vi er ferdig med det
    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 minne og programmet kræsjer. Dette kalles en minnelekkasje. Minnelekkasjer er også i mange tilfeller en sikkerhetsrisiko. Det oppstår lett minnelekkasjer når man bruker dynamisk minne, så beste praksis er å kun bruke det der det er absolutt nødvendig. I eksempelet over er bruk av dynamisk minne helt unødvendig, men det er kun ment som et trivielt eksempel for å demonstrere syntaksen.

Konvensjonen er at den delen av programmet som reserverer minnet «eier» minnet og er derfor også ansvarlig for å frigjøre minnet. I prosedyral kode vil det som regel bety at funksjonen der minnet blir reservert også frigjør minnet, og i objektorientert kode (som bruker klasser) betyr det at man allokterer minne i konstruktøren til et objekt og frigjør det i destruktøren.

For å allokere minne til en tabell gjør man følgende:

```
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**!*

## Introduksjon til klasser med dynamisk allokerede medlemsvariabler

Det oppstår et spesialtilfelle vi må ta hensyn til når vi har dynamisk allokerede 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?

```

Når vi skriver `b = a` kaller vi på `operator =`. Vi har imidlertid ikke definert noen slik operator. Operatoren som blir kalt er da en som kompilatoren lager automatisk. Denne operatoren tar en binær kopi, som vil si at den kopierer medlemsvariabel for medlemsvariabel uten å bry seg om noen av dem er dynamisk allokert.

Dette er et problem fordi den eneste medlemsvariabelen i klassen `Example`, `anInt`, 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 allokerede 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 blokken til if-setningen kommer til å bli kjørt siden vi vet at `a` er 5, og derfor større enn null. Vi kopierer `a` til `b`, `b` skrives ut, og i det vi forlater blokken vil `b` bli destruert. Med andre ord vil `b` sin destruktør 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 allokerede heltallet, vil pekeren i `a` nå være ugyldig.

Hvordan kan vi fikse dette? Det vi har beskrevet over kalles en *grunn kopi* (shallow copy), og er ikke alltid en dårlig løsning. I så fall må man holde nøye orden på hvor mange pekere som peker til det samme minnet, og det er utenfor pensum i dette emnet.

Løsningen er å implementere såkalt *dyp kopiering* (deep copying) i `operator =`. En dyp kopi av en peker kopierer også *minneområdet* som pekeren peker til. Når vi kopierer et objekt vil medlemsvariablene i det nye objektet peke til nye, dynamisk allokerede minneområder, og vi unngår problemene beskrevet over.

## Kopikonstruktør og operator =

For å implementere dyp kopiering for `Example`-klassen, må vi legge til eksplisitte implementasjoner av en *kopikonstruktør* 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, for eksempel ved å skrive `Example b(a);`.

```
Example(const Example &other) : anInt(nullptr) {  
    this->anInt = new int(); // Vi allokterer 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 automatisk opprettes med bruk av kopi-konstruktøren.

Deretter swapper vi alle medlemsvariabler (inkl. pekere til allokert data) mellom de to objektene. Dermed får `this` (venstresiden) tilordnet høyresidens verdier (og allokerter 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;  
}
```