TDT4102
Prosedyre- og objektorientert programmering

Klasser, innkapsling, konstruktører, m.m.

Dagens forelesning

- Klasser og objekter
  - Implementering og deklarering
  - Bruk
  - Konstruktører
  - Initialiseringsliste
  - Klasse som medlemsvariabel
  - Static medlemsvariabler og –funksjoner
  - Const
- STL vector (hvis tid)
- Eksempel på bruk
  - Simulering av iskremkiosk episode 2:
    - Struct, array av struct, class, string, vector? m.m.

Kap. 7

Klasser

- Forenkler samspillet mellom data og funksjonalitet
  - Data er medlemsvariabler
  - Medlemsfunksjonene er den funksjonaliteten vi trenger for disse data'ene

- Vi finner mange nyttige klasser i C++ bibliotekene
  - disse er objekter: instanser av klasser!
  - allerede benyttet cin og cout mye
  - string-klassen i biblioteket <string>

Klasser: eksempel (repetisjon)

- Circle-klassens deklarasjonen

```
class Circle{
private:
    double radius;
public:
    void setRadius(double r);
    double getRadius();
    double getArea();
    double getCircumference();
};
```

- Implementasjon av medlems-funksjoner

```
void Circle::setRadius(double r){
    radius = r;
}
double Circle::getRadius(){
    return radius;
}
double Circle::getArea(){
    return PI * radius * radius;
}
double Circle::getCircumference(){
    return 2 * radius * PI;
}
```

Instansiering og initialisering

- **Instans**iering er å lage et nytt objekt
  - foreløpig har vi gjort dette ved å deklarere variabler av en klassetype
- **Init**ialisering er å sørge for at objekter har en fornuftig startverdi
  - tilsvarende som for initialisering av andre variabler
- I OO-språk brukes "konstruktører" til dette
  - en funksjon som kalles automatisk ved instansiering!
- Typisk brukt til å sette medlemsvariabler til fornuftige startverdier – eller til brukerdefinerte verdier gjennom parametere

Konstruktører (i)


```
class Circle{
private:
    double radius;
public:
    //Konstruktør
    Circle(double r);
```

deklarasjon

implementasjonen

```
//Implementasjon av konstruktøren
Circle::Circle(double r){
    radius = r;
}
```

```
Circle oneCircle(5);
Circle anotherCircle = Circle(10);
```

bruk

Konstruktører (ii)

- Merk! Konstruktører er som vanlige medlemsfunksjoner
- MEN:
  - Har samme navn som klassen
  - Har IKKE returtype
  - Må være public slik at vi kan instansiere objekter
- Vi kan overlagre (overloade) konstruktører
- Dvs. lage konstruktører for forskjellige formål

```
//Konstruktør
Circle();
Circle(double r);
Circle(double r, Color c);
```

Default konstruktør

- Hvis vi IKKE implementerer en konstruktør for en klasse vil kompilatoren automatisk lage en default konstruktør

```
Circle::Circle( ){ }
```

- Med andre ord kan du instansiere objekter selv om du ikke deklarerer en konstruktør for en klasse

```
Circle c;
//eller
Circle c2 = Circle();
```

```
//Men hvorfor kan vi ikke skrive:
Circle c3();
```

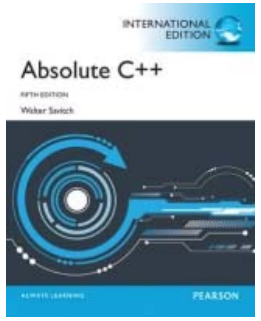
Default konstruktører

- NB! Hvis vi lager en konstruktør med parameter, vil kompilatoren IKKE generere default konstruktør
  - og dermed må du bruke de konstruktørene som er implementert

Inline medlemsfunksjoner

- Vi kan også implementere medlemsfunksjoner direkte i klassedeklarasjonen
- Kalles ”inline” og benyttes normalt kun for svært enkle funksjoner (helst hvis bare en setning)
- Bedre ytelse, fordeler og ulemper, se side 329

```
class Circle{
private:
    double radius;
public:
    //Konstruktør
    Circle(){radius = 0;}
    Circle(double r) {radius = r;}
    void setRadius(double r){radius = r;}
    double getRadius(){return radius;}
    double getArea();
    double getCircumference();
};
```



medlemsfunksjonene implementert som inline

og to som ikke er inline

Konstruktører og bruk av initialiseringliste (i)

- Medlemsvariable kan initialiseres i klasse-definisjonen til bare *en* verdi
  - Den blir en defaultverdi som gjelder all objekter, med mindre man setter den verdien når en instans av klassen (et objekt) konstrueres via en konstruktør
  - Vi ønsker veldig ofte ulike verdier i ulike instanser
  - En konstruktør med initialiseringsliste gjør dette
- Initialiseringliste brukes for å sikre at medlemsvariablene initialiseres riktig før konstruktørens implementasjon utføres
- For klasser som har medlemsvariabler som er av andre klasser, så vil medlemsvariablenes konstruktører kalles først

Syntaks for initialiseringsliste

- Initialiseringsliste spesifiseres i konstruktørens implementasjon
  - etter parameterlista, men før implementasjonsblokka
- Generell syntaks:

```
MinKlasse::MinKlasse() : v1(), v2(), v3() { ..... }
    // Her er v1, v2 og v3 medlemsvariable i klassen
```

- Parametrene i klassens konstruktør kan brukes som argumenter i initialiseringslista

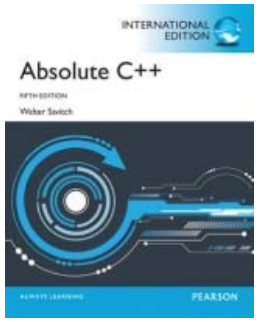
```
MinKlasse::MinKlasse(int x, int y, int z) : v1(x), v2(y), v3(z) { ... }
    // Betyr at v1 = x, v2 = y, v3 = z
```

- Vi kan også initialisere variabler av basistypene på samme måte: `Circle::Circle(int r) : radius(r){}`

Initialiseringsliste (ii), som side 320

```
class Ring{
private:
    Circle inner;
    Circle outer;
public:
    Ring(double o, double i);
};
```

Klasse Ring, som har to objekt av klasse Circle som medlemsvariabler




```
class Circle{
private:
    double radius;
public:
    // 2 Konstruktører
    Circle(){radius = 0;}
    Circle(double r) {radius = r;}
    void setRadius(double r){radius = r;}
    double getRadius(){return radius;}
    double getArea();
    double getCircumference();
};
```

Hvordan sørge for at inner og outer blir instansiert med kall til riktig konstruktør?

For å kalle riktig konstruktør for Circle-variablene i Ring-klassen bruker vi initialiseringsliste

```
Ring::Ring(double o, double i): outer(o), inner(i){
  //Konstruktør med initialiseringsliste
}
```

Eksempel

- Circle.cpp
  - I dag
    - initialisering av Ring-objekt med verdier, tilsvarer læreboka side 320++
  - (Senere ser vi på den biten av koden merket med
    - initialisering av Ring-objekt med to Circle-objekter som parameter)

Administrativt

- Status lærebok
  - 60 eksemplar skal være på vei
- Hjelp i R1, Kollokvie, Øv.foreles, veiledning på sal, forum
- Øvinger
  - Vi leser feedback
  - Gradvis overgang fra stor grad av «hånd-holding» til mer selvstendig programmering og problemløsing
- Spørsmål?

Navnekollisjoner

- Hva om vi bruker samme navn på parameter som vi har brukt til medlemsvariabel?

```
Circle::Circle(double radius){
    radius = radius;
} // ER FEIL
```

- Parametervariabel radius velges først siden man alltid leter opp et navn fra det innerste scope og utover og vi tilordner parameterverdien til parametervariabelen! --- det er ikke hensikten. (vi når ikke objektets medlemsvariabel radius)
- Vi kan skille mellom medlemsvariabel og parameter vha. this

this er en medlemsvariabel alle objekter har – som peker til objektet (seg selv)

```
Circle::Circle(double radius){
    this->radius = radius;
}
```

-> er en operator som kombinerer dereferering med medlemsoperatoren

static variabler (repetisjon)

- Parametre og andre variabler som deklareres og brukes i et funksjonskall eksisterer kun så lenge funksjonen er aktiv
- Av og til ønsker vi å ta vare på verdier fra kall til kall
  - nyttig for å løse spesielle problemer (F.eks. telle opp antall ganger en funksjon er kalt opp)
- Variabler deklarert som static vil initialiseres en gang og deretter beholde sin verdi
  - static er et nøkkelord som vi setter foran datatypen
  - slike nøkkelord kalles generelt for modifikatorer

Bruk av static medlemsvariabler

- Vi har tidligere lært at variabler i funksjoner kan deklareres som static, og at vi kan deklarere konstanter vha. const
- static kan også benyttes for medlemsvariabler
  - en static medlemsvariabel er EN felles variabel for alle instanser
  - hvis vi deklarerer den som const får vi en felles konstant variabel

```
class Circle{
private:
   static int counter;
   double radius;
public:
   static const double PI;
   Circle(double r);
   void setRadius(double r);
   double getRadius();
   double getArea();
   double getCircumference();
};
```

I klasse-deklarasjonen

Deklareres i klassen og initialiseres i implementasjonen (merk at static nøkkelordet ikke skal være med i impl.)

```
int Circle::counter = 0;
const double Circle::PI = 3.14;
```

Funksjonalitet

```
Circle::Circle(double radius){
    counter++;
    this->radius = radius;
}
```

- Deklarerte:
  - Static int counter
  - Counter vil bli inkrementert hver gang konstruktøren kalles
  - Dvs. teller hvor mange objekter som blir opprettet
- En static const medlemsvariabel fungerer som en vanlig konstant, men som er deklarert som en del av klassen
  - kan leses uten at det finnes noen variabel ved å bruke klassenavn som scope

```
cout << Circle::PI << endl;
```

static medlemsfunksjoner

- Også medlemsfunksjoner kan være static
  - Dette gir en litt spesiell funksjonalitet siden det gjør det mulig å lage funksjoner som er uavhengige av instanser
  - Vil i praksis være som andre funksjoner implementert utenfor klassene
  - Men kan i tillegg lese/endre static medlemsvariabler
- Er en del brukt til ”utility” funksjoner som det er logisk å ha som en del av en klasse
  - f.eks. en funksjon som beregner areal bare basert på argumentverdi

```
static double getArea(double r){return PI * r * r;}
```

Mer bruk av const

- const kan brukes sammen med call-by-reference
  - for å spesifisere at en funksjon ikke skal endre på parameterverdien
- For medlemsfunksjoner kan vi bruke const for å spesifisere at en funksjon ikke endrer på noen av medlemsvariablene
  - i dette tilfellet plasseres const etter funksjonsdeklarasjonen (og brukes tilsvarende i implementasjonens header)

```
double getArea() const;
```

Alternativer til arrays:
parameteriserte typer (templates)
Nytt og viktig tema (intro)

- Vanlige arrays er en lavnivå konstruksjon som du må kunne, men i praktisk programmering bruker vi ofte andre ting
- F.eks. <vector>
- Mange andre etterhvert

Vector

- Tabeller (arrays) er ikke bestandig like praktiske å bruke
- En annen løsning er å bruke en datatype som kan endre størrelse ved behov: vector
- Dette er en klasse
  - del av standard template library (STL)
  - template-klasser er en type klasser som ″tilpasser″ seg datatypen de benyttes for

vector<type>

- Er en tabell-lignende konstruksjon som har dynamisk størrelse ("fleksibel tabell")
- Legger til elementer med funksjonskallet push_back(type)
- lese/skrive verdier med indeksoperatoren

```
#include <vector>
```

```
vector<int> myVector = {1, 2, 3};
myVector.push_back(4);
for (int i = 0; i < myVector.size(); i++){
    cout << myVector[i] << endl;
}
```

Vector intro

- Likt med tabeller
  - Kan lagre *en* type data (basistype, klassetype, ...)
- Deklarerering
  - Syntaks: vector<Base_Type>
    - Hakeparanteser < > indikerer template-klasse
    - En hvilken som helst "type" kan være basetype
    - I praksis vil kompilatoren lage en vector-klasse for deg ved kompilering som støtter den spesifiserte datatypen
- Eksempel deklareringer:

```
vector<int> v;

vector<double> d;

vector<Sirkel> sirkler;
```

Bruk av vector

- vector<int> v;
  - "v er vector av typen int"
  - default konstruktør blir kalt automatisk
    - Et tomt vector-objekt blir instansiert
- Legge til elementer: `v.push_back(i);`
- Hente ut elementer: `cout << v[i];`
- Men vi må fortsatt huske å sjekke størrelsen
  - ved hjelp av size() som returnerer antallet elementer i vektoren

```
for (int i = 0; i < v.size(); i ++){
    cout << v[i] << endl;
}
```

array<int, size>

- Er i praksis en vanlig tabell, men pakket inn i en template-klasse
- Bruker indeksoperatoren for å lese/skrive
- Kan "huske" sin egen størrelse

```
#include <array>
```

```
array<int, 4> myArray = {1, 2, 3};
myArray[3] = 100;
for (int i = 0; i < myArray.size(); i++){
    cout << myArray[i] << endl;
}
```

En klasse for lavnivå arrays: array<T, size>

- Er en template-type for arrays av fast størrelse
- Medlems-funksjoner som for vector
- Men ellers lik vanlige arrays

```
#include <iostream>
#include <array>
using namespace std;

int main(){
    //en array<T,size) for int
    array<int, 5> myIntArray = {1,2,3,4,5};

    for (int i = 0; i < myIntArray.size(); i++){
        cout << myIntArray[i] << endl;
    }
    //en array<T,size) for double
    array<double, 3> myDoubleArray = {1.0,2.0,3.0};
    for(double t: myDoubleArray){
        cout << t << endl;
    }
}
```

C++ v11
«range» for løkke

Simulering av iskremkiosk, *episode 2*

- Repeterte raskt
  - Tabell av struct (vist sist)
  - Ble glemt (les selv):
    - Ny versjon av generateStudents
      - Kortere, enklere pga. bruk av <string>

Iskrem_V2.zip

- Nytt (vist raskt)
  - Klasse Event
  - Tabell av pekere til Event objekter
    - Funksjon for å generere «ArrivalEvents» dvs. løkke med kall på new Event med 3 parametere
    - PrintAllEvents som viser innholdet av alle Event objektene som vi har adgang til gjennom denne tabellen av pekere til Event objekter

- Resten av koden ble kjørt, men ikke forklart (vi kommer tilbake til dette)