BOKMÅL

NTNU
Norges teknisk-naturvitenskapelige universitet

Fakultet for informasjonsteknologi, matematikk og elektroteknikk

Institutt for datateknikk og informasjonsvitenskap

Tentativt løsningsforslag
TDT4102 Prosedyre og objektorientert programmering

Fredag 6. juni 2008
Kl. 09.00 – 13.00

NB! Husk at det kan være flere måter å løse en oppgave på!

Sensurdato:
Sensuren faller 3 uker etter eksamen og gjøres deretter kjent på http://studweb.ntnu.no/.

Prosentsatser viser hvor mye hver oppgave teller innen settet.

Merk: Alle programmeringsoppgaver skal besvares med kode i C++.

Lykke til!

Generelt for alle oppgaver

- Merk at ikke alle deloppgaver krever at de foregående er løst, så ikke hopp over de resterende deloppgavene om én blir for vanskelig.
- Der det spesifikt står definer eller deklarer er vi kun interessert i funksjonprototyper eller klassedeklarasjoner. Der det står implementer skal du vise implementasjon av funksjoner.
- Der det spesifikt spørres etter forklaringer og beskrivelser er det et av kravene at dette skal være med.

OPPGAVE 1 (20 %): Funksjoner

a) Implementer funksjonen **`int max (int tab[], int size)`** som returnerer høyeste verdi som finnes i en tabell. Husk at tabellen kan inneholde alle lovlige int-verdier inklusive negative tall. Parameter **`size`** angir størrelsen på tabellen og i denne oppgaven kan du anta at tabellen alltid vil inneholde ett eller flere tall.

```
int max(int tab[], int size){
    int m = tab[0];
    // eller int m = INT_MIN;
    cout << m;
    for (int i = 0; i < size; i++){
    // kan begynne med 1 hvis du bruker [0] (sjekker på size);
        if (m < tab[i]){
            m = tab[i];
        }
    }
    return m;
}
```

Itererer over tabellelementene og tar vare på verdien hvis den er høyere enn det som allerede er lagret i m. Her er det et viktig poeng å initialisere m med en verdi enten fra tabellen (det er også mulig å bruke konstanten INT_MIN som finnes i biblioteket <climits>).

b) Implementer funksjonen **`void printAZ()`** som skriver ut bokstavene A-Z med maks 7 bokstaver på hver linje. Tips: **`int`** og **`char`** er kompatible typer i C++ og de samme operatorer som brukes på int kan brukes på char, for eksempel er det mulig å inkrementere en char for å få tegnet som kommer etter.

```
void printAZ(){
    int counter = 0;
    for (char i = 'A'; i <= 'Z'; i++){
        cout << i << " ";
        if (++counter == 7){
            cout << endl;
            counter = 0;
        }
    }
    cout << endl;
}
```

Merk at det står skriv ut "bokstavene" og at det hintes om om inkrementering (mindre trekk for å skrive ut "ferdige strenger" for eksempel "A B C D E F G")

En enda mer kompakt versjon (uten teller) er:

```
void printAZ(){
    for (char i = 'A'; i <= 'Z'; i++){
        cout << i << " ";
        if ((i - 'A' + 1)  % 7 == 0){
            cout << endl;
        }
    }
    cout << endl;
}
```

c) I brettspill er det ofte regler om hvordan brikker kan flyttes. Implementer en funksjon som sjekker om du flytter diagonalt i et brettspill (fra posisjon "fromx, fromy" til "tox, toy")
`bool isDiagonal (int fromX, int fromY, int toX, int toY)`

isDiagonal(1, 1, 8, 8) → true fordi du her flytter like mange steg til langs x- og y-aksene
isDiagonal(8, 1, 1, 8) → true fordi du her flytter like mange steg til langs x- og y-aksene
isDiagonal(1, 7, 5, 8) → false fordi du ikke flytter like mange steg langs begge akser
I tilfellet *fromX == toX og fromY == toY* skal du returnere false.

Hint: Hva er forholdet mellom endringen i rad og kolonne i et diagonalt flytt? Her kan du også bruke funksjonen **`int abs(int i)`** i tilfelle du trenger absoluttverdien av et tall.

```
bool isDiagonal(int fromX, int fromY, int toX, int toY){
    return ((abs(toX - fromX) == abs(toY - fromY)) &&
    (toX - fromX) != 0);
}
```

d) Hva er overlagring (overloading) og hvorfor er det av og til praktisk å overlagre funksjoner? Hva slags feil/problemer vil du få hvis du prøver å kompilere følgende kode? Forklar!

```
int max(int i, int j);
double max(double i, double j);
double max(double i, int j);
int max(int i, int &j);

int main(){
    int k = max(1,2);
    double l = max(1,2);
    double m = max(1.0, 1);
    double n = max(1,1);
    int o = max(k, k);
}
```

I C++ (og en del andre programmeringsspråk) er det tillatt at det finnes flere funksjoner med samme navn så lenge de har forskjellig parameterliste/returtype.
Dette er ganske praktisk siden vi da slipper å tenke på disse som forskjellige funksjoner siden de "konseptuelt" fremstår som en og samme funksjon med variabel parameterliste (selv om det i praksis likevel er mange funksjoner)

Når overlagring er benyttet bruker kompilatoren parameterlista for å finne ut hvilken av de overlagrede funksjonene som skal benyttes, basert på kompatiblitet med typene funksjonen kalles med. I tilfellet over vil kompilatoren for for den siste linjen i koden ikke klare å finne ut hvilken funksjon som skal kalles og dermed vil du få kompileringsfeil.

OPPGAVE 2 (25 %): Diverse

a) Hva er nødvendig å implementere for at du skal kunne bruke sammenligningsoperatorer for å sammenligne verdier av følgende egendefinerte datatype? Forklar og vis eksempel på implementasjon!

```
struct amount {
    string currency;
    double value;
};
```

Nevn de sammenligningsoperatorer det er naturlig å implementere for denne datatypen.

Her er vi ute etter overlagring av operatorer. Et forenklet eksempel på dette er koden under. (På eksamen er det ikke nødvendig å ha med const.)

```
bool operator ==(const amount& a, const amount &b){
    return ((a.currency == b.currency) && (a.value == b.value));
}
```

I praksis kan man implementere <, >, <=, >=, != og ==, men hvis valuta er forskjellig er det derimot vanskelig å vite om et beløp er > eller < og derfor er det mer naturlig å kun implementere != og ==, eller kanskje bare == (og sjekke om valuta-stringene er like). Det er også mulig å tenke seg en hjelpefunksjon som konverterer mellom forskjellige valuta og da er det fullt mulig å støtte < og > for alle beløp, men det er det ikke nevnt noe om i oppgaven og dermed kan dere ikke anta at det finnes en slik funksjon.

Merk at denne oppgaven ikke viser til neste og er ment som en selvstendig oppgave hvor vi også var ute etter litt refleksjon rund hvilke operatorer det er naturlig å støtte for denne typen.

b) Operatorer kan også kaste unntak. Utvid implementasjonen fra oppgaven over slik at det kastes et unntak av en egendefinert unntakstype hvis du prøver å sammenligne to verdier av forskjellig "currency". Vis kode for hvordan du kan fange akkurat denne spesifikke unntakstypen.

```
//Unntaksklasse og instans
class CurrencyException{};
//operator som kaster unntak
bool operator ==(const amount& a, const amount &b)
throw(CurrencyException)
{
    if (a.currency != b.currency){
        throw CurrencyException();
    }else{
        return (a.value == b.value);
    }
}
//try/catch eksempel
amount a1 = {"NOK", 30.55};
amount a2 = {"EUR", 30.50};

try{
    bool b = (a1 == a2);
}catch(CurrencyException &c){
    cout << "Catching currency exception!" << endl;
}
```

NB! (vanlig feil) Det er selvsagt riv ruskende galt å ha try-catch med i implementasjonen av operatoren. Da vil jo ikke operatoren kaste noe unntak siden den selv vil fange opp unntaket den

c) Forklar kort hva som er forskjellen mellom peker-variabel og "vanlige" variabel?
Rett opp koden under slik at **`x`** etter kall til swap har verdien 2 og **`y`** har verdien 1. Du skal **ikke** endre funksjonsheader for **`swap`**, men kun legge til **`*`** og/eller **`&`** i **`swap`** og/eller i **`main`**.

```
void swap(int* x, int& y){
    int temp = y;
    y = x;
    x = temp;
}

int main(){
    int x = 1;
    int y = 2;
    swap (x, y);
    //nå skal x ha verdien 2 og y ha verdien 1
}
```

Vanlige variabler er navngitte "plassholdere" for verdier. Disse er selvsagt lagret på en spesifikk plass i minnet, men programmerer som om de var verdier. Pekere-variabler er også variabler, men slike variabler inneholder en adresse til en minnelokasjon. Se pensumboka for mer detaljert beskrivelse ;-)
Eller sagt på en annen måte: vanlige variabler er verdier, mens pekere er adresser til verdier.

*Tre endringer er nødvendige. Husk at peker-argumentet x må være en adresse; derfor brukes &x i funksjonskallet. Referanseargumentet y er en verdi-variabel og for å bytte om verdiene må vi derfor bruke y = *x og *x = temp.*

```
void swap(int* x, int& y){
    int temp = y;
    y = *x;
    *x = temp;
}

int main(){
    int x = 1;
    int y = 2;
    swap (&x, y);
    //nå skal y ha verdien 1 og x ha verdien 2
}
```

d) Tenk deg et program som implementerer en telefonbok med informasjon om personer. For å holde informasjon om personer brukes en egendefinert datatype **`Person`** (klasse eller struct). Telefonboken skal kunne inneholde et antall personer som er ukjent ved programmering og det skal være mulig å legge til og slette personer.

- Hvis du ønsker å bruke en vanlig tabell for å lagre personer, hva må/da til for at du kan endre størrelsen på tabellen i kjøretid for eksempel for å øke antallet personer som kan lagres.

Da må vi bruke dynamiske tabeller og lage et program hvor vi lager en ny og større tabell hvis plassbehovet økes (og kopiere over eksisterende data). (Dette bør i praksis kombineres med delvis fylt array: dvs. en tabell hvor vi både kjenner maks-størrelse og hvor mange elementer som er i bruk. Da kan vi øke størrelsen i bolker slik at vi unngår unødvendig mye kopiering.)
*Person *p = new Person[100];*

- Gitt at du ikke vil bruke tabell og heller IKKE skal bruke standardbiblioteket, hva slags egendefinert datastruktur kan du da evt. lage deg for å vedlikeholde en mengde med personer i programmet ditt?

En lenket datastruktur. Noder som er lenket vha. pekere mellom nodene; enkeltlenket liste, dobbeltlenket liste, tre-struktur.

- Hvilke mulige eksisterende typer i standardbiblioteket er egnet for å løse oppgaven over (lagre, slette, legge til personer)?

For eksempel vector, set, map.

Typer som stack og queue er mindre relevante for denne typen applikasjoner!

- Hvorfor kan vi bruke slike typer i standardbiblioteket på egendefinerte datatyper (dvs. typer som ikke var kjent når standardbiblioteket ble implementert)?

Fordi de er implementert som templates. Det vil si klasser eller funksjoner hvor typen(e) som klassen eller funksjonen brukes for er parameteriserte.

OPPGAVE 3 (25 %): Sjakkspill

I denne oppgaven skal du lage klasser som kan brukes i implementasjon av et sjakkspill. Sjakk spilles på et 8x8 brett og i et sjakkspill finnes det 6 forskjellige typer brikker (konge, dronning, tårn, løper, hest, bonde) av to forskjellige farger (svart og hvit). Hvis du ikke kjenner sjakk-spillet finner du en kort forklaring bakerst i eksamensoppgaven, men det du trenger av informasjon er også inkludert i deloppgavene.

I denne oppgaven skal du kun implementere noen utvalgte brikke-typer og fokusere på en begrenset del av funksjonaliteten i spillet.

a) Vis hvordan du kan deklarere en **`enum`**-type kalt Colour for å definere fargeverdiene **`svart`** og **`hvit`**. Forklar kort fordelene med å bruke enumeration (**`enum`**) i stedet for andre måter å representere disse verdiene på (for eksempel string, char, int).

```
enum Colour {BLACK, WHITE};
```

Fordelen er at datatypen Colour nå bare har disse to lovlige verdier; Black og White. Det er med andre ord en måte å kontrollere at vi kun bruker lovlige verdier (i tillegg er det også enklere å huske slike navn). Andre fordeler er økt lesbarhet m.m.

b) I denne oppgaven skal du vise fornuftig bruk av arv, innkapsling og konstruktør(er). Sjakkbrikkene skal implementeres som klasser; en klasse for hver type brikke. Alle brikke-klassene skal være subtyper av klassen **`Piece`**. Implementer klassene **`Piece`**, **`Queen`** (dronning), **`Rook`** (tårn) og **`Bishop`** (løper) slik at du ivaretar følgende funksjonalitet:

- Felles for alle brikker er at de har en medlemsvariabel for farge av typen **`Colour`** (bruk enum beskrevet i deloppgave a).
- Felles for alle brikkene er at fargen til en brikke skal settes av konstruktøren og ikke kunne endres etterpå. For alle brikker skal du kunne kalle medlemsfunksjonen **`getColour()`** for å få returnert verdien for en brikkes farge (svart eller hvit).

```
class Piece{
   private:
      Colour colour;
   public:
      Piece(Colour c): colour(c){};
      //ikke krevd at du har initialiseringsliste her
      Colour getColour(){return colour;};
};

class Queen : public Piece{
   public:
      Queen(Colour c): Piece(c){};
};

class Rook : public Piece{
   public:
      Rook(Colour c): Piece(c){};
};

class Bishop : public Piece{
   public:
      Bishop(Colour c): Piece(c){};
};
```

Her er vi ute etter bruk av private for medlemsvariabel colour, arv, konstruktør og bruk av supertypens konstruktør i initialiseringslista.

Her er det feil å definere nye getColour() for Queen, Rook og Bishop, men konstruktørene arves ikke og må derfor deklareres (og implementeres) for hver klasse. Også viktig å kalle superklassens konstruktør i initialiseringslista.

Husk at Colour er en type (selv om det er en enum) på linje med om du hadde skrevet int colour.

Litt selvkritikk til oppgaven: her var det egentlig litt unødvendig å be om alle klassene siden det ble mye repetisjon, men argumentet for å ha med alle klassene var å tydeliggjøre bruken av arv. I praksis hadde vi oppnådd det samme ved å redusere antallet klasser som skulle implementers til Piece og f.eks , Queen og Rook.

c) Et sjakkbrett består av 8x8 ruter. I hver rute kan det stå en brikke av en hvilken som helst type og farge, men en rute kan også være tom (dvs. når det ikke står en brikke på ruten). I sjakk-verdenen brukes notasjonen ”a1”, ”b2”, ”h5” etc for å angi rute på brettet hvor bokstavene a-h angir kolonne (x-koordinat) og 1-8 angir rad (y-koordinat).

- Deklarer en klasse for et sjakkbrett **`Board`**. Bruk en tabell som kan inneholde pekere til Piece-objekter som medlemsvariabel for brettet.
- Deklarer medlemsfunksjoner som du kan bruke for å oversette fra sjakk-notasjon til verdier som kan brukes som tabellindekser (her er vi først og fremst interessert i funksjonsprototypene).
- Implementer medlemsfunksjonene
    - **`void setPiece(Piece* piece, string position)`**
    - **`Piece* getPiece(string position)`**

Den første brukes for å plassere en brikke i en rute og den siste brukes for å finne ut hvilken brikke som står på en bestemt rute. Her skal du vise hvordan du vil bruke dine egne funksjoner for å oversette fra string-notasjon til tabellindeks.

```
class Board{
private:
    Piece* board[8][8];
public:
    Board();
    static int getX(string position);
    static int getY(string position);
    Piece* getPiece(string pos);
    void setPiece(Piece* piece, string pos);
};


Piece* Board::getPiece(string pos){
    return board[getX(pos)][getY(pos)];
}

void Board::setPiece(Piece* piece, string pos){
    board[getX(pos)][getY(pos)] = piece;
}


// Brettet bør implementeres som [8][8] og IKKE [64] siden board[2][3]
// er forskjellig felt fra board[3][2], mens board[3*2] == board[2*3].
// Velger man [64] må du bruke [(x*8) + y] da vil a1 -> [(0*8) + 0]
// b2 -> [(1*8) + 1] osv… NB! Det er en grunn til at vi kan lage
//flerdimensjonale tabeller i stedet for å måtte regne ut plassen selv!
```

d) Hvordan vil du håndtere tomme ruter i deloppgave c)? Dvs. hva mener du getPiece(string) skal/kan returnere hvis posisjon-argumentet angir en rute hvor det ikke står noen brikke?

En grei løsning er å bruke NULL (eller 0). Da er det enkelt å sjekke om det står en brikke på et felt. Dette kan for eksempel implementeres slik at Board sin konstruktør setter alle felter til å være NULL før brikkene settes opp. Når en brikke flyttes slik at feltet igjen blir tomt bør startposisjon igjen settes til NULL. Hvis denne mekanismen implementeres er det ingen grunn til å spesialhåndtere tomme plasser i funksjonene over da det allerede er ivaretatt.

NB! Det er egentlig ikke noen god ide å kaste et unntak hvis en kaller getPiece på et felt som er tomt. Unntakshåndtering bør brukes på litt mer alvorlige feil. Riktignok er det en mulig løsning, men det ville forutsette at man sjekket om feltet var tomt før hvert kall til getPiece (noe som er unødvendig tungvint)……

OPPGAVE 4 (30 %): Sjakkspill forts.

I et sjakkspill er det forskjellige regler for hvordan brikkene kan flyttes. Et tårn kan bare flytte langs en rad eller en kolonne. En løper kan bare flyttes diagonalt, kun hesten har lov til å hoppe over andre brikker etc. Gå ut i fra at følgende hjelpefunksjoner allerede er implementert for **`Board`**-klassen. Disse hjelpefunksjoner skal du seinere bruke for teste på om flytting av en brikke er lovlig.

- **`bool isStraight(string from, string to)`**
  Denne funksjonen returnerer true hvis et trekk fra **`from`** til **`to`** er langs en rad eller langs en kolonne.
- **`bool isDiagonal(string from, string to)`**
  Denne funksjonen returnerer true hvis et trekk fra **`from`** til **`to`** går diagonalt, altså på skrå over brettet.
- **`bool isOccupiedBetween(string from, string to)`**
  Denne funksjonen returnerer true hvis det finnes brikker på plassene mellom from og to og kan brukes både for rette og diagonale trekk.
- **`bool isEmpty(string to)`** sjekker om feltet **`to`** er tomt eller ikke.
- **`string getPosition(Piece* piece)`**
  Denne funksjonen returnerer posisjonen en gitt brikke har på brettet

a) Hva betyr det hvis vi deklarerer en medlemsfunksjon som **`static`**? Hvorfor kan det være hensiktsmessig å deklarere noen av hjelpefunksjonene beskrevet over som **`static?`** Hvilke(n) av funksjonene over kan du ikke deklarere som **`static`**?

Medlemsfunksjoner som er static bruker ikke medlemsvariabler (de kan bruke medlemsvariabler som er static, men ikke variabler som er ikke-static). Slike funksjoner kan derfor kalles uavhengig av instans, men med klassenavnet som scope for eksempel Board::isDiagonal("a2", "b3");

Fordelen med å ha slike funksjoner er at de kan kalles uavhengig av om vi har ett objekt av denne typen eller ikke.

Av funksjonene over er det:

`bool isStraight(string from, string to)`

`bool isDiagonal(string from, string to)`

som kan være static. De øvrige funksjoner returnerer verdier som kan være forskjellig avhengig av "tilstanden" i et Board-objekt. Her vil det også være nødvendig å definere funksjonene som gjør om fra string-notasjon til koordinater som static.
Dekket på side 293 i pensumboka (selv om det ikke er noen innførsel for static i lærebokas indeks).

b) Utvid programmet ditt med kode slik at du i brikkenes medlemsfunksjoner kan kalle **`Board`**-objektet's medlemsfunksjoner. I oppgave 3 var det bare kobling fra brett til brikke, men nå skal du implementere disse klassene slik at det også er en kobling/referanse fra en

brikke til brettet den står på.

I prinsippet må man da sørge for at Piece-objektene får en referanse til Board-objektet. Dette kan gjøres vha. en set-Metode eller i konstruktøren som vist under:

```
class Piece{
    private:
        Colour colour;
        Board* b;
    public:
        Piece(Colour c, Board* b): colour(c), b(b){};
        Colour getColour(){return colour;};
};

class Queen : public Piece{
    public:
        Queen(Colour c, Board* b): Piece(c, b){};
};
```

Tilsvarende for de andre sub-typene også. Bruker man en set-metode i Piece-klassen er det ikke nødvendig å endre subklassene, da denne automatisk blir tilgjengelig via arv.

NB! Nøkkelordet friend brukes bare for å overstyre private modifikatoren og har ingenting med denne oppgaven å gjøre, med mindre man ønsker at Piece-objekter skal kunne få tilgang til private-medlemsvariabler eller medlemsfunksjoner i Board-objekter – da må du også huske på å ha med en Board-variabel i Piece-klassen.

NB!! Det er heller ikke noe poeng å arve fra Board for da vil hver brikke få hvert sitt brett!

c) Denne deloppgaven kan løses ved å ta i bruk funksjoner som allerede er beskrevet i oppgave 3 og 4. Du skal vise bruk av virtuelle funksjoner og arv slik at de forskjellige brikke-klassene implementerer forskjellige reglene for hva som er lovlige trekk, men du skal også passe på at samme kode ikke blir duplisert. Implementer følgende for brikkene:

- **`bool canTake(string to)`** sjekker om du har lov til å ta brikken på feltet **`to`** (i praksis er dette bare å sjekke at fargen til brikken som evt. står på to-feltet er den motsatt av fargen til brikken som flyttes).
- **`bool canMove(string to)`** sjekker om veien frem til **`to`**-feltet-brikken er lovlig, for eksempel kreves det for tårnet at det ikke hoppes over brikker og at flytt-operasjonen går langs en rad eller langs en kolonne.
- **`void move(string to)`** sjekker at trekket er lovlig og utfører trekket.

Følgende regler gjelder for de brikkene du skal implementere disse funksjonene for:

- En dronning (**`Queen`**) kan flyttes horisontalt, vertikalt eller diagonalt.
- Et tårn (**`Rook`**) kan flyttes horisontalt og vertikalt (men ikke diagonalt).
- En løper (**`Bishop`**) kan flyttes diagonalt (men ikke horisontalt eller vertikalt).
- Ingen av disse brikkene kan hoppe over andre brikker.

Feltet det flyttes til må være tomt eller det må stå en brikke av motsatt farge der.

```
// Board-klassen er ikke del av løsningen, men tatt med for at
// det skal bli litt mer oversiktelig
// Her er det viktig å ha vist hvilken funksjon som er virtual

class Board{
private:
    (Piece*) board[8][8];
public:
    Board();
    static int getX(string position);
    static int getY(string position);
    static bool isStraight(string from, string to);
    static bool isDiagonal(string from, string to);
    bool isOccupiedBetween(string from, string to);
    bool isEmpty(string pos);
    string getPosition(Piece* piece);
    Piece* getPiece(string pos);
    void setPiece(Piece *piece, string pos);
};


class Piece{
    private:
        Colour colour;
        Board* b;
    public:
        Piece(Colour c, Board *b): colour(c), b(b){};
        Colour getColour(){return colour;};
        Board* getBoard(){return b;};
        bool canTake(string to);
        virtual bool canMove(string to) = 0;
        void move(string to);
};

FORTS. NESTE SIDE
```

```
class Queen : public Piece{
   public:
      Queen(Colour c, Board* b): Piece(c, b){};
      virtual bool canMove(string to);
};


class Rook : public Piece{
   public:
      Rook(Colour c, Board* b): Piece(c, b){};
      virtual bool canMove(string to);
};

class Bishop : public Piece{
   public:
      Bishop(Colour c, Board* b): Piece(c, b){};
      virtual bool canMove(string to);
};

bool Piece::canTake(string to){
   if (!b->isEmpty(to)) &&
   (b->getPiece(to)->getColour() == colour)){
      return false;
   }
   return true;
}

void Piece::move(string to) {
   if (canMove(to)){
      if (b->isEmpty(to)){
         b->setPiece(this,to);
      }else if(canTake(to)){
         delete b->getPiece(to);
         b->setPiece(this,to);
      }
   }
   //  her burde vi også satt feltet vi flytter fra til NULL
}

bool Queen::canMove(string to){
   string from = getBoard()->getPosition(this);
   return (Board::isDiagonal(from, to) || Board::isStraight(from, to))
      && !getBoard()->isOccupiedBetween(from, to);
}

bool Rook::canMove(string to){
   string from = getBoard()->getPosition(this);
   return Board::isStraight(from, to) &&
   !getBoard()->isOccupiedBetween(from, to);
;
}

bool Bishop::canMove(string to){
   string from = getBoard()->getPosition(this);
   return Board::isDiagonal(from, to) &&
   !getBoard()->isOccupiedBetween(from, to);
}
```

Litt mer om spillet sjakk

Sjakk spilles på et 8x8 brett, hvor rutene angis med bokstav a-h for kolonnen og tall 1-8 for raden. Nederste venstre hjørne har koordinatene ”a1” og motsatt hjørne har koordinatene ”h8”. En rute på brettet kan være tom eller inneholde en hvit eller sort brikke.

Det er 6 forskjellige typer brikker, med hver sine regler for hvordan de flytter. Generelt kan en brikke enten flytte til et tomt felt eller til et felt med en brikke med motsatt farge, heretter kalt en motstanderbrikke. Det siste kalles å slå og brikken som flyttes vil da erstatte motstanderbrikken.

- *Bonden* flytter 1 rute forover og slår motstandere 1 rute diagonalt forover. Merk at ”forover” er oppover for hvit og nedover for svart.
- *Springeren* (hest) flytter 2 ruter i én retning og 1 rute i retning vinkelrett på første, f.eks. 2 ruter frem og 1 til venstre eller 1 rute bakover og 2 til høyre. I motsetning til de andre brikketypene kan en springer hoppe over andre brikker.
- *Løperen* (biskop) flytter 1 eller flere ruter diagonalt i en av 4 retninger og kan ikke hoppe over andre brikker.
- *Tårnet* flytter 1 eller flere ruter rett frem eller til siden i en av 4 retninger og kan ikke hoppe over andre brikker.
- *Dronningen* flytter 1 eller flere ruter rett frem, til siden eller diagonalt i en av 8 retninger og kan ikke hoppe over andre brikker.
- *Kongen* flytter 1 rute rett frem, til siden eller diagonalt i en av 8 retninger. Dersom kongen står slik at den kan slås av motstanderen i neste trekk, så er den *sjakk*.

a8 b8 c8 d8 e8 f8 g8 h8 a7 b7 c7 d7 e7 f7 g7 h7 a6 b6 c6 d6 e6 f6 g6 h6 a5 b5 c5 d5 e5 f5 g5 h5 a4 b4 c4 d4 e4 f4 g4 h4 a3 b3 c3 d3 e3 f3 g3 h3 a2 b2 c2 d2 e2 f2 g2 h2 a1 b1 c1 d1 e1 f1 g1 h1	Figuren til venstre viser koordinat-systemet. Figuren til høyre viser startoppstillingen, med 1 rad bønder og 1 rad med hhv. tårn, springer, løper, dronning, konge, løper, springer og tårn, for hver spiller. De hvite bøndene flytter oppover og de sorte bøndene nedover.