

NTNU
Norges teknisk-naturvitenskapelige
universitet

Fakultet for informasjonsteknologi,
matematikk og elektroteknikk

Institutt for datateknikk
og informasjonsvitenskap



EKSAMEN I FAG
TDT4102 Prosedyre og objektorientert programmering

Fredag 6. juni 2008
Kl. 09.00 – 13.00

Faglig kontakt under eksamen:

Trond Aalberg, tlf (735) 9 79 52 / 976 31 088

Tillatte hjelpemidler:

- Absolute C++, Savitch
- C++ Pocket Reference

Sensurdato:

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

Prosentsetser 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 klassesdeklarasjoner. 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.
- 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 operatorene som brukes på `int` kan brukes på `char`, for eksempel er det mulig å inkrementere en `char` for å få tegnet som kommer etter.
- 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, 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.

- d) Hva er overlaging (overloading) og hvorfor er det av og til praktisk å overlagre funksjoner? Hva slags feil/problemer vil du få hvis du prøver å compilere 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);
}
```

OPPGAVE 2 (25 %): Diverse

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

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

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

- b) Operatører 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.
- c) Forklar kort hva som er forskjellen mellom peker-variable og "vanlige" variable? 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
}
```

- 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.
 - 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?
 - Hvilke mulige eksisterende typer i standardbiblioteket er egnet for å løse oppgaven over (lagre, slette, legge til personer)?
 - Hvorfor kan vi bruke slike typer i standardbiblioteket på egendefinerte datatyper (dvs. typer som ikke var kjent når standardbiblioteket ble implementert)?

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 deklare 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).
- 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).
- 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.
- 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?

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 deklare noen av hjelpefunksjonene beskrevet over som **static**? Hvilke(n) av funksjonene over kan du ikke deklare som **static**?
- 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.
- 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.

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	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.	8	
a7	b7	c7	d7	e7	f7	g7	h7		7	
a6	b6	c6	d6	e6	f6	g6	h6		6	
a5	b5	c5	d5	e5	f5	g5	h5		5	
a4	b4	c4	d4	e4	f4	g4	h4		4	
a3	b3	c3	d3	e3	f3	g3	h3		3	
a2	b2	c2	d2	e2	f2	g2	h2		2	
a1	b1	c1	d1	e1	f1	g1	h1		1	
									a b c d e f g h	