NTNU
Institutt for datateknikk
og informasjonsvitenskap

Eksamensoppgave i

TDT4102 - Prosedyre- og objektorientert programmering

Lørdag 6. juni 2009

Kontaktperson under eksamen: Trond Aalberg (97631088)

Eksamensoppgaven er utarbeidet av Trond Aalberg og kvalitetssikret av Guttorm Sindre og Hallvard Trætteberg

Språkform: Bokmål

Tillatte hjelpmidler: Walter Savitch, Absolute C++ eller Lyle Loudon, C++ Pocket Reference

Sensurfrist: Mandag 29 juni.

Generell intro

Les gjennom oppgaveteksten nøye og finn ut hva det spørres om. Noen av oppgavene har lengre forklarende tekst, men dette er for å gi mest mulig presis beskrivelse av hva du skal gjøre.

All kode skal være C++.

Dersom du mener at opplysninger mangler i en oppgaveformulering, gjør kort rede for de antagelser og forutsetninger som du finner det nødvendig å gjøre. Hver enkelt oppgave er ikke ment å være mer krevende enn det som er beskrevet.

Selv om vi i enkelte oppgaver ber om en funksjon, kan du lage hjelpefunksjoner der du finner at dette er formålstjenelig (f.eks. fordi det gjør det enklere å programmere eller gjør koden mer lesbar).

Oppgavesettet er arbeidskrevende og det er ikke foreventet at alle skal klare alle oppgaver innen tidsfristen. Disponer tiden fornuftig!

Selv om oppgave 2 og 3 omhandler samme tema er det mulig å løse de fleste deloppgaver uten at du har løst de andre oppgavene. Oppgavene er først og fremst relatert til hverandre ved at det er beskrevet funksjoner i en oppgave som er nødvendig for å kunne løse en annen. Du bør derfor lese alle oppgaver selv om du ikke svarer på alle.

Oppgavene teller med den andelen som er angitt i prosent. Den prosentvise uttellingen for hver oppgave kan likevel bli justert ved sensur. De enkelte deloppgaver kan også bli tillagt forskjellig vekt.

Oppgave 1: Litt av hvert - grunnleggende programmering (15%)

a) Hvilke verdier skrives ut for a, b og c i følgede kode:

```
int a = 0, b = 0, c = 0;

b = 5;
a = b++;
cout << "1: a = " << a << endl;

b = 5;
a = --b;
cout << "2: a = " << a << endl;

a = 6;
b = 5;
c = a/b;
cout << "3: c = " << c << endl;
```

b) Implementer en rekursiv funksjon **`void printReverse(char *str)`** som skriver ut en C-streng tegn for tegn **baklengs** til **`cout`.** Med rekursiv mener vi en funksjon som kaller seg selv. For hvert kall skal det bare være ett tegn som skrives ut, men siden du bruker rekursjon vil hele strengen til slutt bli skrevet ut. Oppgaven skal løses uten bruk av andre funksjoner.

```
char *s = "This is a test";
printReverse(s);

// Dette skal gi utskriften: "tset a si sihT"
```

c) Ta utgangspunkt i en klasse **`xclass`** og en overlagret sammenligningsoperator **`<`** som kan brukes for å finne ut om en instans av **`xclass`** er "mindre enn" en annen instans. Vis hvordan denne operatoren sammen med de boolske operatorene **`&&`**, **`||`** og **`!`** kan brukes for å teste om to objekter (**`a`** og **`b`**) av typen **`xclass`** er like. Her er vi ute etter et sammensatt uttrykk som evalueres til **`true`** hvis **`a`** og **`b`** er like, og **`false`** hvis de ikke er like.

Oppgave 2: SUDOKU-funksjoner (40%)

Sudoku består i å skrive inn tall i et rutebrett bestående av 9x9 ruter. Brettet er videre delt inn i ni mindre regioner på 3x3 ruter. I en sudokuopgave er noen av rutene ferdig utfylte og den som løser en sudokuoppgave må fylle inn de tomme rutene slik at tallene fra 1 - 9 brukes en gang i hver region, hver rad og hver kolonne. Du finner en mer detaljert beskrivelse i appendikset til eksamensoppgaven.

I denne oppgaven skal du implementere funksjoner som kan brukes for å løse sudokuoppgaver automatisk. Alle funksjoner som er beskrevet kan være relevante å benytte i andre deloppgaver.

VIKTIGE TIPS: Et sudokubrett/oppgave kan enkelt representeres med en todimensjonal tabell av heltall f.eks. **`int board[9][9].`** For å gjøre programmeringen litt enklere kan du skrive funksjoner som baserer seg på at størrelsen av sudokubrettet bestandig er 9x9 og du kan adressere rutene ved hjelp av vanlige tabellindekser **`board[0-8][0-8]`**. I praksis spiller det liten rolle hvilken indeks du velger som rad og hvilken du velger som kolonne, men en praktisk konvensjon er å bruke **`board[row][column]`** og du kan se for deg at **`board[0][0]`** er øverste venstre hjørne.

a) En rute som er fylt inn i et sudokubrett skal ha en verdi mellom 1 og 9, men i en sudoku-oppgave er det også tomme ruter som en funksjon må være i stand til å finne ut at er tomme.
   - Forklar hvordan du kan representere tomme ruter internt i programmet gitt at variabelen for et brett er **`int board[9][9]`** og ruter som er fylt inn har verdiene 1-9.

b) Implementer funksjoner for å lese/skrive innholdet i et sudokubrett fra/til en tekstfil. Du bestemmer selv hvordan informasjonen skal lagres på fil (inkl. hva som skal lagres for tomme ruter). Det skal være mulig å lese inn filer som er skrevet ut vha. av save-funksjonen og det skal være enkelt å lage en sudoku-oppgavefil for hånd (ved hjelp av en teksteditor).
   - **`void save(int board[][9], char* filename)`**
   - **`void load(int board[][9], char* filename)`**
   - Gi et eksempel på hvordan en sudokuoppgave vil være lagret på fil.

c) Implementer funksjonen:
   - **`bool empty(int board[][9], int row, int column)`**
     som sjekker om en spesifisert rute i et sudokubrett er tom - det vil si at det ennå ikke er fylt inn noen verdi i denne ruten.

d) Implementer **en** av funksjonene:
   (velg selv hvilken du vil implementere - de er ganske like):
   - **`bool inColumn(int board[][9], int column, int value)`**
     Funksjonen skal returnere **`true`** hvis **`value`** finnes fra før i kolonnen.
   - **`bool inRow(int board[][9], int row, int value)`**
     Funksjonen skal returnere **`true`** hvis **`value`** finnes fra før i raden.

e) Implementer funksjonene:

- **`int regionIndex(int index)`**
  Denne funksjonen brukes til å finne ut hvilken 3x3 region en spesifikk rute tilhører ved at den kan regne ut indeksene som denne 3x3 regionen starter med. Funksjonen kan brukes på begge dimensjoner. Eksempel på en bruk av funksjonen:
  `board[regionIndex(0)][regionIndex(1)]` *er det samme som* **`board[0][0]`**
  `board[regionIndex(0)][regionIndex(5)]` *er det samme som* **`board[0][3]`**
  `board[regionIndex(5)][regionIndex(7)]` *er det samme som* **`board[3][6]`**
  `board[regionIndex(7)][regionIndex(8)]` *er det samme som* **`board[6][6]`**
  `osv.`
- **`bool inRegion(int board[][9], int row, int column, int value)`**
  Funksjonen skal returnere **`true`** hvis **`value`** finnes fra før i en 3x3 region av brettet som har som har **`row`** og **`column`** som startposisjon. Eksempler på bruk: **`inRegion(board, 0, 0, 5)`** som sjekker om tallet 5 finnes fra før i øverste venstre 3x3 region.

f) Implementer funksjonen:

- **`bool canBePlacedIn(int board[][9], int row, int column, int value)`**
  Funksjonen brukes for å finne ut om et tall kan plasseres i en spesifikk rute uten å bryte med de grunnleggende reglene i sudoku. Funksjonen skal returnere **`true`** hvis ruten er tom og hvis value ikke finnes fra før i raden, kolonnen eller i 3x3 regionen som ruten er del av.

g) I sudoku bruker man forskjellige strategier for å finne hvilke tall som skal skrives i hvilke ruter. Ved hjelp av funksjonene fra oppgavene over er det mulig å implementere noen av de enkleste løsningsstrategiene. I denne oppgaven er vi interessert i hvordan du kan løse et litt komplekst problem ved å lage/bruke funksjoner som løser hver sin del av problemet.

   Implementer **en** av funksjonene under (velg selv hvilken du vil implementere):

- **`int applySinglePossibleValueStrategy(int board[][9])`**
  Funksjonen finner og fyller inn alle tomme ruter i en sudokuoppgave hvor det kun er ett tall som er mulig fordi alle de andre tallene finnes fra før i raden, kolonnen eller 3x3 regionen. Funksjonen returnerer antallet ruter som er fylt inn. Se appendiks for illustrasjon.
- **`int applyOnlyPossiblePlaceStrategy(int board[][9])`**
  Funksjonen finner og fyller inn alle tomme ruter i en sudokuoppgave hvor det kun er ett tall som er mulig fordi tallet ikke kan plasseres annet sted i raden, kolonnen eller 3x3 regionen. Funksjonen returnerer antallet ruter som er fylt inn. Se appendiks for illustrasjon.

h) Ved å bruke strategiene over vekselvis er det mulig å løse enkle og middels vanskelige sudokuoppgaver. Implementer en funksjon som leser inn en oppgave fra fil, løser denne så langt det er mulig med de strategiene som er beskrevet i oppgaven over og skriver ut løsningen til en annen fil:

- **`void solve(char *infilename, char *outfilename)`**

Oppgave 3: Objektorientert SUDOKU (25%)

I denne oppgaven skal du lage en objektorientert utgave av sudokuspillet som du lagde funksjoner for i oppgave 2.

a) Deklarer en klasse **`sudoku`** for en sudoku-oppgave. Klassen skal ha alle funksjoner og variabler fra oppgave 2 som medlemmer. Du trenger IKKE å implementere funksjonene fra oppgave 2 på nytt, men du må vise/forklare hva som vil være forskjellig. Eventuelle nye funksjoner må implementeres. I denne oppgaven er det viktig å tenke objektorientert (innkapsling, public/private, konstruktør(er) etc).

b) Hvilke(n) av funksjonene i oppgave 2 kan deklareres som static medlemsfunksjon - og hvorfor?

c) Ved lesing/skriving fra/til fil er det flere ting som kan gå galt. For både lesing og skriving kan filnavnet/stien være feil eller det kan være andre grunner til at du ikke får åpnet en fil. Ved lesing kan det hende at fila inneholder ugyldige tall, for mange tall etc. I denne oppgaven skal du implementere en versjon av **`save`** og **`load`** (fra oppgave 2) hvor du tar i bruk unntaksmekanismen. Det viktigste i denne oppgaven er at du viser bruk av unntaksmekanismen - ikke at du håndterer alle tenkelige feil.

   - Lag 2 egendefinerte unntakstyper for feil som henholdsvis er relatert til åpning av ei navngitt fil og feil som er relatert til ugyldig innhold i filer som leses. Begge unntakstypene skal arve fra bibliotekklassen **`exception`**. Denne har en konstruktør som tar inn en tekststreng (C-streng) som parameter. Denne tekststrengen kan du få returnert ved å kalle **`what()`**-funksjonen som exception-klassen implementerer.
   - Vis hvordan du i **`save-`** og **`load`**-funksjonene kaster unntak av disse typene ved feil. Unntakene som kastes skal inneholde informasjon om feilen (en tekststreng).
   - Vis hvordan du kan fange opp og skille mellom unntak av disse typene (f.eks. i main) og får skrevet ut informasjon om unntakene.

d) Hvis du har løst sudokuoppgaver har du sikkert erfart hvor irriterende det er å gjøre feil underveis. Hvis du husker de siste tallene du har fylt inn kan du viske ut og prøve på nytt, men ofte har du glemt sekvensen av tall som er fylt inn. I denne oppgaven skal du implementere en funksjon for å skrive inn tall i en sudokuoppgave og en funksjon for å slette tall som er skrevet inn. Funksjonene skal f.eks. kunne brukes ved manuell løsing av en oppgave.

   - Definer datatype(r) og/eller medlemsvariable(r) som er nødvendige for at **`sudoku`**-klassen skal kunne huske hvilke ruter som er fylt inn og i hvilken rekkefølge de er fylt inn.
   - Implementer en medlemsfunksjon
     `void enter(int row, int column, int value)`
     som kan brukes til å fylle inn ruter i en sudoku-oppgave (for "manuell" løsing av en sudokuoppgave - f.eks. basert på input som er lest inn med en annen funksjon).
   - Implementer en medlemsfunksjon
     `void undo()`
     som kan brukes til å angre/fjerne det siste tallet som er skrevet inn. Undo skal kunne kalles flere ganger etter hverandre for å fjerne en sekvens av tall i motsatt rekkefølge av hvordan de ble satt inn.

Oppgave 4: Personer i et familietre (20%)

a) Deklarer og implementer klassen **`Person`**. Objekter av denne typen skal brukes til å danne et familietre hvor hver person har en mor og en far samt ingen eller flere egne barn som personen selv er forelder til. Du skal deklarere og implementere de medlemsvariabler, medlemsfunksjoner etc. som er nødvendige for å kunne kjøre koden under nøyaktig slik den er skrevet - og slik at den produserer utskrift som er lik den som vises under. Her er det viktig å tenke objektorientert.
Merk at det bare er **`setMother`** og **`setFather`** som brukes for å sette opp slektstreet og at **`setMother`** kalles to ganger med samme argument uten at dette fører til duplisering blant barna denne personen har.

b) Hva vil skje hvis **`setMother`** kalles på samme objekt 2 ganger, men med forskjellig mor som argument (basert på hvordan du har implementert klassen)? Beskriv ved hjelp av tekst eller vis med kode hvordan du ville valgt å håndtere dette (vi er ute etter hvordan du ville valgt å løse/håndtere dette mulige problemet).

```
int main(){
   Person *me = new Person("Per", "Hansen");
   Person *mybrother = new Person("Arne", "Hansen");
   Person *myhalfbrother = new Person("Jakob", "Jonsen");
   Person *mysister = new Person("Turid", "Hansen");
   Person *mymother = new Person("Johanne", "Hansen");
   Person *mymother = new Person("Johanne", "Hansen");
   Person *myfather = new Person("Severin", "Hansen");
   Person *mychild = new Person("Anders", "Hansen");

   me->setFather(myfather);
   me->setMother(mymother);
   mychild->setFather(me);
   mybrother->setFather(myfather);
   mybrother->setMother(mymother);
   myhalfbrother->setMother(mymother);
   mysister->setFather(myfather);
   mysister->setMother(mymother);

   cout << *me << endl<< endl;
   cout << *myfather << endl<< endl;
   cout << *mymother << endl<< endl;
}
```

Utskriften som dette produserer:

Name: Per Hansen
Father: Severin Hansen
Mother: Johanne Hansen
Children: Anders

Name: Severin Hansen
Children: Arne, Per, Turid

Name: Johanne Hansen
Children: Arne, Jakob, Per, Turid

Appendiks

Beskrivelse av sudoku

Sudoku består i å skrive inn tall i et rutebrett bestående av 9x9 ruter. Brettet er videre delt inn i ni mindre regioner på 3x3 ruter (vist med fete streker i eksempelet under). I en sudokuopgave er noen av rutene ferdig utfylte og den som løser en sudokuoppgave må fylle inn de tomme rutene slik at tallene fra 1 - 9 brukes en gang i hver region, hver rad og hver kolonne.

En sudokuoppgave

						4		**8**
				1	**9**			**2**
			4	**3**			**9**	**5**
		6		**5**				**9**
	1	**5**	**9**				**6**	
	3				**8**	**5**		**4**
1					**6**			
		2		**4**				
4	**6**	**8**	**5**		**3**			

Løsningen på oppgaven

9	*2*	*3*	*6*	*7*	*5*	**4**	*1*	**8**
5	*7*	*4*	*8*	**1**	**9**	*6*	*3*	**2**
6	*8*	*1*	**4**	**3**	*2*	*7*	**9**	**5**
2	*4*	**6**	*3*	**5**	*7*	*1*	*8*	**9**
8	**1**	**5**	**9**	*2*	*4*	*3*	**6**	*7*
7	**3**	*9*	*1*	*6*	**8**	**5**	*2*	**4**
1	*5*	*7*	*2*	*8*	**6**	*9*	*4*	*3*
3	*9*	**2**	*7*	**4**	*1*	*8*	*5*	*6*
4	**6**	**8**	**5**	*9*	**3**	*2*	*7*	*1*

Reglene

9	2	3	6	7	5	4	1	8
5	7	4	8	1	9	6	3	2
6	8	1	4	3	2	7	9	5
2	4	6	3	5	7	1	8	9
8	1	5	9	2	4	3	6	7
7	3	9	1	6	8	5	2	4
1	5	7	2	8	6	9	4	3
3	9	2	7	4	1	8	5	6
4	6	8	5	9	3	2	7	1

Alle tallene 1-9 skal finnes i hver kolonne

9	2	3	6	7	5	4	1	8
5	7	4	8	1	9	6	3	2
6	8	1	4	3	2	7	9	5
2	4	6	3	5	7	1	8	9
8	1	5	9	2	4	3	6	7
7	3	9	1	6	8	5	2	4
1	5	7	2	8	6	9	4	3
3	9	2	7	4	1	8	5	6
4	6	8	5	9	3	2	7	1

Alle tallene 1-9 skal finnes i hver rad

9	2	3	6	7	5	4	1	8
5	7	4	8	1	9	6	3	2
6	8	1	4	3	2	7	9	5
2	4	6	3	5	7	1	8	9
8	1	5	9	2	4	3	6	7
7	3	9	1	6	8	5	2	4
1	5	7	2	8	6	9	4	3
3	9	2	7	4	1	8	5	6
4	6	8	5	9	3	2	7	1

Alle tallene 1-9 skal finnes i hver 3x3 region

Løsningsstrategiene som er omtalt i oppgave 2 g)

Eksempel på SinglePossibleValueStrategy

				7		4		8
				1	9		3	2
			4	3			9	5
		6		5				9
	1	5	9				6	
	3				8	5		4
1					6			
		2		4				
4	6	8	5		3			

				7		4	1	8
				1	9		3	2
			4	3			9	5
		6		5				9
	1	5	9				6	
	3				8	5		4
1					6			
		2		4				
4	6	8	5		3			

Her kan vi bare plassere tallet 1. 2,3,4,5,8,9 finnes i regionen fra før, 6 finnes i kolonnen og 7 finnes i raden.

Eksempel på OnlyPossiblePlaceStrategy

				7		4	1	8
				1	9		3	2
			4	3			9	5
		6		5				9
	1	5	9				6	
	3				8	5		4
1					6			
		2		4				
4	6	8	5		3			

				7		4	1	8
				1	9		3	2
			4	3			9	5
		6		5				9
	1	5	9				6	
	3				8	5		4
1					6		4	
		2		4				
4	6	8	5		3			

Her kan vi bare plassere tallet 4 fordi det ikke kan plasseres noe annet sted i denne regionen. De grå radene og kolonnene inneholder allerede tallet 4 og dermed står vi igjen med bare en mulig rute for tallet 4 i denne regionen.