NTNU
Institutt for datateknikk
og informasjonsvitenskap

TENTATIVT LØSNINGSFORSLAG

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 << "5: c = " << c << endl;
```

```
1: a = 5
2: a = 4
3: c = 1
```

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"

//Test på termineringstegnet '\0' viser at du kjenner C-strenger,
//rekursivt kall med s + 1 viser at du kjenner pekere og pekeraritmetikk,
//utskift etter det rekursive kallet viser at du har forstått rekursjon.
void printReverse(char* s){
   if (*s != '\0'){
      printReverse(s + 1);
      cout << *s; //eller cout << s[0];
   }
}
```

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.

```
//Her var det ment at du skulle bruke < sammen med &&, ||, ! (trekk for
å bruke andre operatorer - og du måtte ikke bruke alle)

//Vi ber bare om et sammensatt uttrykk:
!(a < b) && !(b < a)
eller
!(a < b || b < a)
```

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.

Det er behov for en fast verdi for tomme ruter og i praksis kan du velge en hvilken som helst verdi utenom 1-9 så lenge den samme verdien brukes konsekvent. 0 er brukt i dette løsningsforslaget.

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.

Gitt at 0 representerer tomme ruter kan vi f.eks lagre informasjonen som en sekvens av tall adskilt med mellomrom (eller tab/linjeskift). Dette gjør det enkelt å redigere en sudokuoppgave og samtidig kan vi enkelt lese og skrive vha. << og >>

```
0  0  0  0  0  0  4  0  8
0  0  0  0  1  9  0  3  2
0  0  0  4  3  0  0  9  5
+ 6 linjer til
```

For lesing og skriving er det viktig å ha med:
riktige klasser (ifstream for lesing, ofstream for skriving) og bruk av open og close
testing på om fila lar seg åpne (helt greit å ikke teste på feil filinnhold)
riktig koding av uskrift/innlesing
hvis << og >> benyttes må du skrive ut space og linjeskift
oppgaven kan også løses ved å skrive/lese linjer eller enkelttegn

```
void load1(int board[][9], char *filename){
   //iterer over cellene og leser en og en verdi fra fil
   ifstream inputfile(filename);
   if (!inputfile){
      //noe er feil med fila og vi avslutter uten å lese
      return;
   }
   for (int i = 0;i < 9; i++){
      for (int j = 0;j < 9; j++){
         // vi antar at fila inneholder 9x9 tall
         inputfile >> board[i][j];
      }
   }
   inputfile.close();
}

void load2(int board[][9], char *filename){
   //hvis du heller vi ha en ytre løkke som tester på slutten av fila
   //må du beregne tabell-indeksene selv (mindre kode, men det er sikkert ikke så mange
   //som kom på denne måten for å beregne riktige tabellindekser).
   ifstream inputfile(filename);
   if (!inputfile){
      //noe er feil med fila og vi avslutter uten å lese
      return;
   }
   int k = 0;
   while (inputfile){
      inputfile >> board[k/9][k%9];
      k++;
   }
   inputfile.close();
}

void save(int board[][9], char *filename){
   ofstream outputfile(filename);
   if(!outputfile){
      //noe er feil med fila og vi avslutter uten å skrive ut noe
      return;
   }
   for (int i = 0;i < 9; i++){
      for (int j = 0;j < 9; j++){
         outputfile << board[i][j] << " "; //skriver ut tallet + space mellom tallene
      }
      outputfile << endl; // ny linje for hver rad
   }
   outputfile.close();
}
```

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.

```
bool empty(int board[][9], int row, int column){
   return (board[row][column] == 0);
}

//vi kunne også laget oss en konstant for 0 kalt EMPTY
```

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.

```
bool inRow(int board[][9], int row, int value){
   for (int column = 0; column < 9; column++){
      if (board[row][column] == value){
         return true;
      }
   }
   return false;
}

bool inColumn(int board[][9], int column, int value){
   for (int row = 0; row < 9; row++){
      if (board[row][column] == value){
         return true;
      }
   }
   return false;
}
```

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)]` -> **`board[0][0]`**
  `board[regionIndex(0)][regionIndex(5)]` -> **`board[0][3]`**
  `board[regionIndex(5)][regionIndex(7)]` -> **`board[3][6]`**
  `board[regionIndex(7)][regionIndex(8)]` -> **`board[6][6]`**
  `osv.`

```
//Pilene her var ment for å vise at det på venstre side var det samme som det på høyre
side, men det var litt uheldig å bruke pil i noe som så ut som kodelinjer .....
//Selve funksjonen vi er ute etter er svært enkel, og beskrivelsen var en forklaring på
hva du kan bruke den til; hvilken region tilhører ruten board[7][8]? jo den tilhører ned-
erste høyre region (subtabell av tabellen) som "starter" med board[6][6]

int regionIndex(int index){
   return (index / 3) * 3;
   // eller return index - (index%3);
}
```

- **`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.

```
bool inRegion(int board[][9], int row, int column, int value){
   for (int i = row; i < (row + 3); i++){
      for (int j = column; j < (column + 3); j++){
         if (board[i][j] == value){
            return true;
         }
      }
   }
   return false;
}

bool inRegion2(int board[][9], int row, int column, int value){
//med implisitt håndtering av regionindekser
   for (int i = regionIndex(row); i < (regionIndex(row) + 3); i++){
      for (int j = regionIndex(column); j < (regionIndex(column) + 3); j++){
         if (board[i][j] == value){
            return true;
         }
      }
   }
   return false;
}

// Det er kanskje noen som lurte litt på hvordan disse funksjonene skulle virke/brukes,
men dette ble forhåpentligvis litt mer tydelig i neste funksjon hvor vi har bruk for å
sjekke om verdien vi ønsker å plassere i en spesifikk rute kan plasseres i denne regionen.

//Det er også fritt fram hvordan du velger å bruke regionIndex sammen med inRegion.
//I koden sjekker vi bare om verdien finnes i en (hvilken som helst) 3x3 region av tabellen
//og vi antar at den kalles med verdier som ikke gjør at vi prøver å lese utover tabellens
//størrelse
```

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.

```
//Her er vi interessert i hvordan du ville kombinere funksjonene fra tidligere deloppgaver
for å teste om en verdi lovlig kan plasseres i en spesifikk rute

bool canBePlacedIn(int board[][9], int row, int column, int value){
   return empty(board, row, column) &&
         (!inRow(board, row, value)) &&
         (!inColumn(board, column, value)) &&
         (!inRegion(board, regionIndex(row), regionIndex(column), value));
}

eller med implisitt håndtering av regionindeksene og bruk av inRegion2

bool canBePlacedIn(int board[][9], int row, int column, int value){
   return empty(board, row, column) &&
         (!inRow(board, row, value)) &&
         (!inColumn(board, column, value)) &&
         (!inRegion2(board, row, column, value));
}
```

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.

```
I denne oppgaven er vi interessert i hvordan du kan løse et litt mer komplekst problem
ved hjelp av funksjonene vi har laget så langt. Her er det en god ide å lage seg en
hjelpfunksjon for å unngå for mange nestede løkker og uoversiktelig kode.

int onlyPossibleNumber(int board[][9], int row, int column){
   //finner den eneste verdien som lovlig kan plasseres i en rute
   //returnerer tallet hvis det finnes bare ett lovlig tall, ellers returneres 0
   int number = 0; //stores the last found number
   int count = 0;  //counts numbers found
   for (int i = 1; i < 10; i++){ //testing for values 1-9
      if (canBePlacedIn(board, row, column, i)){
            count++; //teller lovlige tall som den finner
            number = i; //setter number til sist lovlige tall som er funnet
      }
   }
   if (count == 1){
      return number; //bare ett lovlig tall - returne dette
   }else{
      return 0; // null eller mange lovlige tall, returner 0
   }
}

int applySinglePossibleValueStrategy(int board[][9]){
   int count = 0; //teller ruter som er fylt inn
   for (int row = 0; row < 9; row++){
      for (int column = 0; column < 9; column++){
         int solution = 0;
         if ((solution = onlyPossibleNumber(board, row, column)) != 0){
            // hvis solution er != 0 så er det bare en lovlig verdi og vi kan fylle inn
            count++; //oppdaterer teller
            board[row][column] = solution; // setter verdien
         }
      }
   }
   return count;
}
```

- **`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.

```
//Denne strategien er litt mer kompleks enn den forrige, men til gjengjeld er den kanskje
litt mer forståelig. I praksis kommer du langt med en hjelpfunksjon som tester om et tall
er mulig andre steder i en region - samme type hjelpefunksjon kan lages for å test om
tallet kan plasseres annet sted i rad eller kolonne, men det holder at du har laget en
slik funksjon og vist riktig tankegang.

bool possibleElsewhereInRegion(int board[][9], int row, int column, int value){
   for (int i = regionIndex(row); i < (regionIndex(row) + 3); i++){
      for (int j = regionIndex(column); j < (regionIndex(column) + 3); j++){
         if ( ( i != row || j != column) && canBePlacedIn(board, i, j, value)){
            return true; //alternativ plass er funnet
         }
      }
   }
   return false;
}

bool possibleElsewhereInRow(int board[][9], int row, int column, int value){
   for (int j = 0; j < 9; j++){
      if ((j != column) && canBePlacedIn(board, row, j, value)){
         return true; //alternativ plass er funnet
      }
   }
   return false;
}

bool possibleElsewhereInColumn(int board[][9], int row, int column, int value){
   for (int i = 0; i < 9; i++){
      if ((i != row) && canBePlacedIn(board, i, column, value)){
         return true; //alternativ plass er funnet
      }
   }
   return false;
}

int applyOnlyPossiblePlaceStrategy(int board[][9]){
   int count = 0; //count the number of values inserted
   for (int value = 1; value <= 9; value++){
      for (int row = 0; row < 9; row++){
         for (int column = 0; column < 9; column++){
            if (canBePlacedIn(board, row, column, value)
               //et litt tricky uttrykk her, men det viktigste er å vise
               //hvilke funksjoner du ville ha kombinert
               && (!possibleElsewhereInRegion(board, row, column, value)
               || !possibleElsewhereInRow(board, row, column, value)
               || !possibleElsewhereInColumn(board, row, column, value))){
               board[row][column] = value;
               count++;
            }
         }
      }
   }
   return count;
```

```
}
```

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)`**

```
//Her er det stort sett bruk for en løkke hvor du anvender begge strategier og summerer
antallet oppdateringer disse gjør. Avsluttes når count er 0.

void solve(char *infilename, char *outfilename){
   int board[9][9];
   load(board, infilename); //leser inn en oppgave fra fil
   int count = 1; //positive value to be pass the first evaluation
   while (count > 0){
      count = 0;
      count += applySinglePossibleValueStrategy(board);
      count += applyOnlyPossiblePlaceStrategy(board);
   }
   save(board, outfilename); //skriver ut løsningen til en annen fil
}
```

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).

```
class sudoku{
private:
   int board[9][9];
   bool empty(int row, int column);
   bool inRow(int row, int value);
   bool inColumn(int column, int value);
   static int  regionIndex(int row);
   bool inRegion(int row, int column, int value);
   bool canBePlacedIn(int row, int column, int v);
   int  onlyPossibleNumber(int row, int column);
   int  applySinglePossibleValueStrategy();
   bool possibleElsewhereInRegion(int row, int column, int value);
   bool possibleElsewhereInRow(int row, int column, int value);
   bool possibleElsewhereInColumn(int row, int column, int value);
   int  applyOnlyPossiblePlaceStrategy();
public:
   sudoku(char* filename); //lager et brett-objekt med innhold fra en fil
   sudoku(); //for å lage en tom oppgave
   void save(char *filename);
   void load(char *filename);
   void solve(char *infilename, char *outfilename);
   void solve(char *outfilename); //i tilfellet vi har lest inn med konstruktøren
};

//Viktig forskjell er at board nå er medlemsvariabel og skal utelates fra parameterlisten!
//Vi trenger også en eller to konstruktør, f.eks. en konstruktør som ikke har paramenter
//og som initialieserer alle ruter til tomme og/eller en konstruktør som tar et filnavn
//som parameter og leser inn
//Generelt er det opp til deg å vurdere hvilke funksjoner som kan være private/public,men
//konstruktør og f.eks. save, load, solve bør minimum være public, board MÅ være private

sudoku::sudoku(){
   for (int row = 0; row < 9; i++){
      for (int column = 0; column < 9; column++){
         board[row][j] = 0;
      }
   }
}

sudoku::sudoku(char *filename){
   load2(filename);
};
```

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

```
static int  regionIndex(int index);
//Fordi den ikke bruker noen medlemsvariabel
```

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 bibliotekssklassen **`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.

```
//her er du selvsagt full mulighet til å korte inn på koden som du skriver så lenge du
viser unntakstypene og hvordan unntakene kastes

class FileError: public exception{
public:
   FileError(char* error):exception(error){};
};

class FileContentError: public exception{
public:
   FileContentError(char* error):exception(error){};
};

void sudoku::load(char *filename){
   //itererer over cellene og leser en og en verdi fra fil
   ifstream inputfile(filename);
   if (!inputfile){ // eller (inputfile.fail())
      //noe er feil med fila og vi kaster unntak
      throw FileError("Feil ved åpning av inputfil!");
   }
   for (int i = 0;i < 9; i++){
      for (int j = 0;j < 9; j++){
         // vi antar at fila inneholder 9x9 tall
         int temp;
         inputfile >> temp;
         if (temp < 0 || temp > 9){
            throw FileContentError("Feil innhold i fil");
         }
         board[i][j] = temp;
      }
   }
   inputfile.close();
}
```

```
void sudoku::load2(char *filename){
   ifstream inputfile(filename);
   if (!inputfile){
      //noe er feil med fila og vi avslutter uten å lese
      throw FileError("Feil ved åpning av inputfil!");
   }
   int k = 0;
   while (inputfile){
      int temp;
      inputfile >> temp;
      if (temp < 0 || temp > 9){
         throw FileContentError("Feil innhold i fil");
      }
      board[k/9][k%9] = temp;
      k++;
   }
   inputfile.close();
}

void sudoku::save(char *filename){
   ofstream outputfile(filename);
   if(!outputfile){
      //noe er feil med fila og vi kaster et unntak
      throw FileError("Feil ved åpning av outputfil!");
   }
   for (int i = 0;i < 9; i++){
      for (int j = 0;j < 9; j++){
         outputfile << board[i][j] << " "; //skriver ut tallet + space mellom tallene
      }
      outputfile << endl; // ny linje for hver rad
   }
   outputfile.close();
}
```

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.

```
//Lager en egen klasse for informasjon om cellen
class cell{
//dropper innkapsling her siden vi bare skal lese og sette enkle verdier
public:
   cell(int row, int column):row(row), column(column){};
   int row;
   int column;
};

//I sudoku-klassen legger vi til denne variabelen (en stakk er grei, men vi kunne også
brukt andre datatyper så lenge vi kan legge til og fjerne fra slutten)

stack<cell> history;
```

- 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).

```
void sudoku::enter(int row, int column, int value){
   board[row][column] = value;
   history.push(cell(row, column)); //lagrer i en stack
}
```

- 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.

```
void sudoku::undo(){
   if (!history.empty()){ //sjekker om stakken er tom for sikkerhets skyld
      cell c = history.top(); //leser øverste element
      history.pop(); //fjerner øverste element
      board[c.row][c.column] = 0;
      //setter ruten til 0 for å indikere at den nå er tom
   }
};
```

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.

```
//Viktige elementer: pekere som medlemsvariabler for mother og father, liste, set eller
vector (alle med Person *) for children. Hvis du ikke bruker set må du implementere
unikhetstesting selv.

class Person;

class Person{
private:
   string firstname;
   string lastname;
   Person *father;
   Person *mother;
   set<Person *> children;
public:
   Person(string first, string last);
   void setFather(Person *father);
   void setMother(Person *mother);
   string getName();
   friend ostream& operator<<(ostream &o, Person p);
};

Person::Person(string first, string last): firstname(first), lastname(last){
   father = NULL;
   mother = NULL;
}

void Person::setFather(Person *person){
   this->father = person;
   person->children.insert(this);
   //siden også barn er Person-typer trenger vi ikke set-funksjoner
}

//samme som for setFather og det er ikke nødvendig å ha impl. begge
void Person::setMother(Person *person){
   this->mother = person;
   person->children.insert(this); *
}

string Person::getName(){
   return lastname + string(", ") + firstname;
}
```

```
//Denne er mye mer omstendig enn dere er forventet å lage
//Det som er viktig er at dere skjønner at vi trenger å overlagre <<
ostream& operator<<(ostream &o, Person p){
   o << "Name: " << p.getName() << endl;
   if (p.father != NULL){
      o << "Father: " << p.father->getName() << endl;
   }
   if (p.mother != NULL){
      o << "Mother: " << p.mother->getName() << endl;
   }
   if (p.children.size() > 0){
      int count = 0;
      int last = p.children.size();
      set<Person *>::iterator it;
      o << "Children: ";
      for (it = p.children.begin(); it != p.children.end(); it++){
         o << (*it)->firstname;
         if (++count < last){
            o << ", ";
         }
      }
   }else{
      o << "No children" << endl;
   }

   return o;
}
```

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).

```
//Slik koden er over ender vi da opp med at samme person er "barn av to mødre"
Alternative løsninger er:
* Ikke gjør noe, men la overlat problemet til den som bruker klassen - banal men grei nok
løsning så lenge den er veldokumentert
* Forhindre at dette gjøres f.eks. ved å kaste unntak, eller ha en funksjon hvor slike
kall ikke får noen effekt,  men da trenger vi f.eks. andre funksjoner for å kunne redi-
gere i strukturen
* Lag en funksjon som håndterer dette ryddig (se under)
//Generelt honorerer vi velbegrunnede løsninger

Rydde opp i strukturen kan f.eks. gjøres med
if (mother != NULL){
   finn this blant mother sine children og fjern denne fra children
}
mother = newmother;
oppdater mother sine barn med this

faktisk kode (men denne bruker find og erase funksjoner som ikke er pensum):
void Person::setMother(Person *person){
   if (mother != NULL){
      set<Person *>::iterator it = mother->children.find(this);
      mother->children.erase(it);
   }
   this->mother = person;
   mother->children.insert(this);
}
```

```
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);
   me->setMother(mymother); //Denne linja manglet i oppgaven
   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

Til slutt: Ja det var ganske mange som klarte å gjøre hele oppgaven. Det var også mulig å få en bra karakter (B) selv om du ikke rakk over alle oppgavene.

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

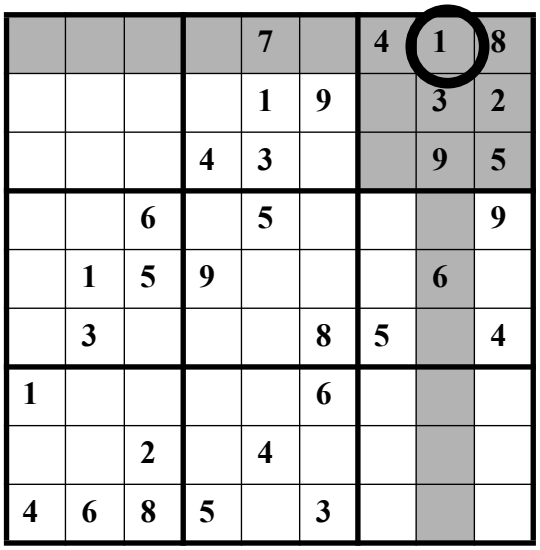
9	2	3	6	7	5	4	1	8
5	7	4	8	1	9	6	3	2
6	8	1	4	3	2	7	9	5
2	4	6	3	5	7	1	8	9
8	1	5	9	2	4	3	6	7
7	3	9	1	6	8	5	2	4
1	5	7	2	8	6	9	4	3
3	9	2	7	4	1	8	5	6
4	6	8	5	9	3	2	7	1

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

Løsningsstrategiene som er omtalt i oppgave 2 g)

Eksempel på SinglePossibleValueStrategy

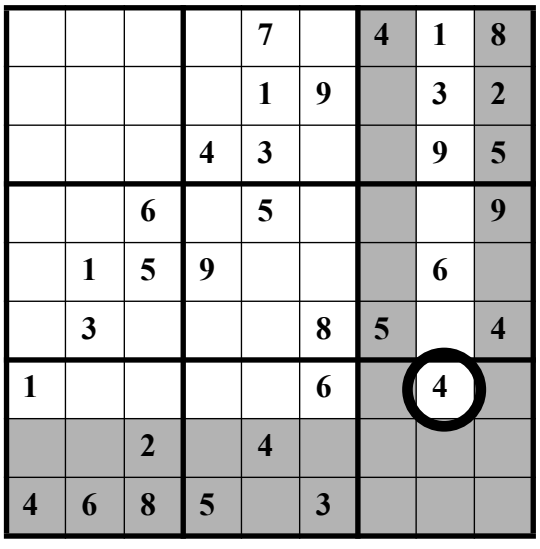
				7		4		8
				1	9		3	2
			4	3			9	5
		6		5				9
	1	5	9				6	
	3				8	5		4
1					6			
		2		4				
4	6	8	5		3			



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			



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.