NTNU
Institutt for datateknikk
og informasjonsvitenskap

Kontinuasjonseksamen i

TDT4102 - Prosedyre- og objektorientert programmering

TENTATIVT LØSNINGSFORSLAG

Fredag 17 august 2012, 09:00

Kontaktperson under eksamen: Trond Aalberg (97631088)

Eksamensoppgaven er utarbeidet av Trond Aalberg

Språkform: Bokmål

Tillatte hjelpmidler (hjelpemiddelkode C):
Walter Savitch, Absolute C++ eller Lyle Loudon, C++ Pocket Reference.
Bestemt, enkel kalkulator tillatt.

Sensurfrist: Fredag 7 september.

Generell introduksjon

Les gjennom oppgavetekstene nøye. Noen av oppgavene har lengre tekst, men dette er for å gi kontekst, introduksjon og eksempler til oppgaven.

Når det står "implementer" eller "lag" skal du skrive en implementasjon. Hvis det står "vis" eller "forklar" står du fritt i hvordan du svarer, men bruk enkle kodelinjer og/eller korte forklaringer og vær kort og presis. Dersom du mener at opplysninger mangler i en oppgaveformulering, gjør kort rede for de antagelser og forutsetninger som du finner nødvendig.

Hver enkelt oppgave er ikke ment å være mer omfattende enn det som er beskrevet. Noen oppgaver fokuserer bare på enkeltfunksjoner og da er det utelukkende denne funksjonen som er tema. Andre oppgaver er "oppskriftsbasert" og vi spør etter forskjellige deler av en klasse, samarbeidende klasser eller et program. Du kan velge selv om du vil løse dette trinnvis ved å ta del for del, eller om du vil lage en samlet implementasjon. Sørg for at det går tydelig frem hvilke spørsmål du har svart på hvor i koden din.

Det er ikke viktig å huske helt korrekt syntaks for biblioteksfunksjoner. Oppgaven krever ikke kjennskap til andre klasser og funksjoner enn de du har blitt kjent med i øvingsopplegget. All kode skal være i C++.

Oppgavene teller med den andelen som er angitt i prosent. Den prosentvise uttellingen for hver oppgave kan/vil likevel bli justert ved sensur basert på hvordan oppgavene har fungert. De enkelte deloppgaver kan også bli tillagt forskjellig vekt.

NB! I denne denne versjonen av eksamensoppgaven har jeg rettet opp noen mindre feil/uklarheter som var i den opprinnelige oppgaven. Alle korreksjoner ble annonsert under eksamen.

Oppgave 1: Oppvarming (20%)

a) Implementer en funksjon **`double mean(int arr[], int size)`** som regner ut det aritmetiske gjennomsnittet av verdiene i en tabellen **`arr`**. Det aritmetiske gjennomsnittet finner du ved å dele summen av alle verdiene med antallet verdier. Merk at resultatet skal bli et flyttall selv om inputverdiene til funksjonen er heltall. Parameteren **`size`** er størrelsen på tabellen (antallet verdier i tabellen).

Eksempel: For verdiene {2,2,2,2,4,4,4,4} blir det aritmetiske gjennomsnittet 24 / 8 = 3.0
For verdiene {2, 3, 4, 5} blir det aritmetiske gjennomsnittet 14 / 4 = 3.5

```
double mean(int arr[], int size){
  int sum = 0;
  for (int i = 0; i < size; i++){
    sum += arr[i];
  }
  return sum / (double) size;
  //eller sum / static_cast<double>(size)
}
Hvis begge operandene er heltall (int) er det viktig å caste den ene til
flyttall slik at vi får flyttallsdivisjon. Heltallsdivisjon som etterpå
konverteres til flyttall og enten returneres direkte eller mellomlagres
i en flyttallsvariabel blir ikke riktig. Det holder å bruke c-style cas-
ting, men du kan også bruke sum / static_cast<int>(size). Noen har også
deklarert sum som double og det er en helt grei løsning i denne oppgaven.
```

b) I statistikk er median den verdien som ligger i midten av en sortert serie med verdier. Hvis antallet verdier er et oddetall, er medianen den midterste verdien. Hvis mengden av verdier er et partall er medianen gjennomsnittet av de to midterste verdiene. Implementer en funksjon **`double median(int arr[], int size)`** som finner medianen for verdiene i den sorterte tabellen **`arr`** (du trenger med andre ord ikke bekymre deg om sorteringen).

Eksempel: For verdiene {1, 1, 3, 4, 8} blir medianen 3.0 siden dette er tallet som ligger i midten. For tallene {1, 1, 3, 4, 5, 8} blir medianen 3.5 siden dette er gjennomsnittet av de to verdiene som ligger i midten (3 og 4).

```
double median(int arr[], int size){
  if (size % 2 == 1){ //oddetall
     return  arr[size / 2]; //baserer oss på impl. konv. av returverdien
  }else{ //partall
    return (arr[size / 2] + arr[(size / 2) - 1]) / 2.0;
  }
}
//Her er trikset å bruke modulus-operatoren for å sjekke om det er partall
//eller oddetall. Deler vi size med på 2 ved oddetall får vi riktig indeks
//eksemplevis er 9 / 2 = 4. Flyttallsdivisjon oppnår vi ved å bruke 2.0
```

c) Implementer en funksjon **`mode(int arr[], int size)`** som finner typetallet for verdiene i den sorterte tabellen arr. Typetallet er det tallet som forekommer oftest i en mengde av tall og for en tallrekke kan det være flere typetall. Du må selv bestemme hva som er en egnet returtype i denne deloppgaven (eller annen teknikk for å få svaret returnert).

Eksempel: For verdiene {1, 1, 2, 2, 3, 3, 3, 4, 5, 6} er typetallet 3 siden det er denne verdien som forekommer flest ganger. For verdiene {1, 1, 2, 2, 3, 3, 3, 4, 4, 4} er typetallene 3 og 4 siden begge disse verdiene forekommer like mange ganger.

Tips: Her er det selvsagt fritt frem å bruke klasser og funksjoner i C++ biblioteket.

```
//To varianter - begge basert på bruk av map<int, int>
vector<int> mode1(int arr[], int size){
  //eller vi kan ha
  //void mode(int arr[], int size, vector<int> &result)
  map<int, int> freq; //vi lager en frekvenstabell (med map)
  for (int i = 0; i < size; i++){
    freq[arr[i]]++;
  }
  //her er en løsning som er grei å kode, men ikke nødvendigvis effektiv
  //vi finner max frekvensverdi først og leter opp nøklene etterpå
  int max = 0;
  for (map<int,int>::iterator it = freq.begin(); it != freq.end(); ++it){
    if (it->second > max){
      max = it->second;
    }
  }
  vector<int> result;
  for (map<int,int>::iterator it = freq.begin(); it != freq.end(); ++it){
    if (it->second == max){
      result.push_back(it->first);
    }
  }
  return result;
}

vector<int> mode2(int arr[], int size){ //en alternativ løsning
  map<int, int> freq; //vi lager en frekvenstabell (med map)
  for (int i = 0; i < size; i++){
    freq[arr[i]]++;
  }
  //Det er også mulig å gjøre dette med bare en iterasjon gjennom map'en.
  int max = 0;
  vector<int> result;
  for (map<int,int>::iterator it = freq.begin(); it != freq.end(); ++it){
    if (it->second == max){
      result.push_back(it->first);
    }else if(it->second > max) {
      result.clear();
      max = it->second;
      result.push_back(it->first);
    }
  }
  return result;
}

//Noen har brukt int* som returtype. Det er i teorien mulig å returnere
en peker til et dynamisk allokert array av tall, men hvordan skal du gi
beskjed om antallet verdier i tabellen?
```

Oppgave 2: Morro med objekter og pekere (30%)

Klassen **Person** under skal brukes for å lage et "nettverk av venner". Hver person kan ha en eller flere andre personer som venn (implementert ved at hvert Person-objekt har et set av pekere til andre Personer). Når Person **a** legger til Person **b** som venn ved hjelp av medlemsfunksjonen **connect**, er det meningen at også **b** sin venneliste oppdateres automatisk med en peker til **a**. Koden i **main** er et eksempel på bruken av funksjonen og skal ikke endres.

a) Hva er galt med implementasjonen av **connect**-funksjonen? Hvordan kan du implementere **connect** slik at den fungerer etter intensjonen?

```
class Person{
  private:
    string name;
    set<Person *> friends;
    set<string *> msgs;
  public:
    Person(const string &name);
    void post(string msg);
    void receive(string *);
    void connect(Person *p){
      friends.insert(p);
      p->connect(this);
    };
};
```

```
int main(){
  Person *a = new person("Thea");
  Person *b = new Person("Tor");
  Person *c = new Person("Per");
  a->connect(b);
  a->connect(c);
  b->connect(c);
  a->post("er på konten");
  b->post("jeg også");
  c->post("ikke jeg");
}
//Nå skal alle være venner med alle
//og alle mottar alle meldinger som
//blir sendt
```

Det blir en uendelig løkke av funksjonskall og programmet vil kræsje til slutt på grunn av stack-overflow! I stedet for å gjøre et kall tilbake med connect kan vi like gjerne legge inn this i p sin friendsliste direkte (noe som er mulig siden personer har lese/skrive adgang til andre person-objekters medlemsvariabler selv om disse er private. public/private gjelder bare på klasse-nivå

```
void connect(Person *p){
  friends.insert(p);
  p->friends.insert(this); //i stedet for p->connect(this);
};
```

```
//Dette er den enkleste løsningen, men da var også mange andre kurante
løsninger som i praksis avbryter rekursjonen inkl. å teste om vennelista
inneholder p før en setter inn osv.
```

b) Implementer funksjonen **post** som brukes til å lage og poste meldinger til nettverket av venner og funksjonen **receive** som brukes for å motta meldinger fra venner. Kort beskrevet skal post-implementasjonen instansiere en dynamisk allokert string med innholdet fra msg-parameteren og iterere over alle vennene og gjøre funksjonskallet receive på disse. Meldinger som en person har skrevet selv eller mottatt fra andre lagres i variabelen **msgs**.

NB! Vi har brukt set<string *> msgs; for enkelthets skyld. Egentlig burde vi vel brukt list eller vector slik at vi fikk lagret meldingene i den rekkefølgen de blir lagt til (og ikke får de sortert på peker slik tilfellet er med set).

```
void Person::post(const string &s){
    string *msg = new string("(" + name + ") " + s);
    receive(msg); //eller msgs.insert(msg);
    set<Person *>::iterator it;
    for (it = friends.begin(); it != friends.end(); ++it){
        Person *p = *it;
        p->receive(msg);
    }
}

void Person::receive(string *s){
    msgs.insert(s);
}
```

c) Hva skjer med dette nettverket av pekere hvis du sletter en person? Forklar eller vis hvordan du vil implementere en destruktør for **`Person`**-klassen.

Her blir det mange pekere til samme objekt både for personer og meldinger. Det største problemet er egentlig alle pekerne til personobjektene. Sletter vi en person i kjøretid ender vi opp med vennelister som inneholder ygyldige pekere. Dette kan vi f.eks. rydde opp i ved å be alle venner av en person om å fjerne sine pekere før en person skal slettes.

```
void Person::removeFriend(Person *p){
    friends.erase(p);
}
```

Her fjerner vi pekeren p fra friends. Merk at det dynamisk allokerte objektet ikke blir frigitt (vi sletter pekeren, ikke objektet det pekes til)

I destruktøren iterer vi over alle venner, og gjør kall til removeFriends. (PS! Vi kan jo også slette pekere i vennenes lister uten å gå via en removeFriend-funksjon)

```
set<Person *>::iterator it;
for (it = friends.begin(); it != friends.end(); ++it){
  (*it)->removeFriend(this);
}
```

Videre kan vi rydde opp i meldingene som er opprettet. Egentlig er dette et litt annet problem. Selv om en person slettes vil det ikke medføre at meldingene frigies og dermed er alle pekere til disse fortsatt gyldige. Problemet er at vi kan ende opp med et program som bruker mer og mer minne etter hvert som medlinger opprettes. En løsning er at personer har en variabel som inneholder alle meldinger opprettet av denne personen. Da kan vi frigi disse når personen slettes (i destruktøren), men da må vi også sørge for at andre personers pekere til meldingsorbjektet også fjernes (eller settes til NULL).

d) Implementer funksjonalitet i Person-klassen slik at det ikke er mulig å ha flere Person-instanser med samme navn. Hvis du prøver å lage et nytt objekt mens det finnes et annet objekt som har samme personnavn, skal det kastes et unntak av typen **`NameInUse`**.

Hvis du har behov for å finne en verdi i en container kan du bruke find i <algorithm>som litt forenklet ser slik ut: **`iterator find(iterator first, iterator last, T value)`** *Den returnerer en iterator til verdien som er funnet, eller last hvis den ikke finner verdien. Det finnes også en egen find-funksjon for set:* **`iterator set::find(T value)`**. *Denne returnerer samme iterator som set::end() hvis verdien ikke finnes.*

Her er løsningen å bruke en static medlemsvariabel. Vi kan for eksempel ha en medlemsvariabel deklarert med `static set<string> names`; Siden dette er en static medlemsvariabel må den initialiseres utenfor klassen.

Denne variabelen vil da være delt mellom alle instanser og i konstruktøren kan vi teste på om navnet finnes i names og kaste unntak hvis det finnes fra før. Siden det skal være mulig å gjenbruke navn på objekter som er slettet, må vi også fjerne navn i destruktøren.

```
//Legg til static set<string> names;
class Person{
private:
    static set<string> names;
};

set<string> Person::names;

Person::Person(const string &name){
  if (find(names.begin(), names.end(), name) != names.end()){
  //eller if (names.find(name) != names.end())
     throw NameInUseException();
  }else{
     this->name = name;
     names.insert(name);
  }
}

Person::~Person(){
  names.erase(name);
}
```

Oppgave 3: Et litt større prosjekt (50%)

I denne oppgaven skal du implementere noen klasser og funksjoner som er inspirert av spillet Scrabble /Wordfeud. Dette er et spill hvor spillerne etter tur lager ord med bokstavbrikker på et brett med 15x15 felter. Detaljene og reglene du trenger å kjenne til blir beskrevet etter hvert i deloppgavene, og vi har utelatt det som ikke er relevant for oppgavene du skal løse. Du finner også en beskrivelse av spillet i vedlegget til oppgaven.

a) Implementer en klasse kalt **`Tile`** for bokstavbrikkene. En brikke har en bokstavverdi og en heltallsverdi som begge skal være innkapslet. Bokstav og tallverdi skal settes med konstruktøren og skal kunne leses, men ikke endres etter instansiering. Bruk initialiseringsliste for å sette verdiene. Eksempel på bruk av konstruktøren finner du i koden under.

```
//Her er klassen med inline implementasjoner av get-funksjonene og kon-
struktøren. I tillegg har vi deklarer operatorene i 3b) som friends av
klassen.

class Tile{
public:
    char letter;
    int value;
public:
    char getLetter() {return letter;}
    int getValue() {return value;}
    Tile(char letter, int value): letter(letter), value(value){}
    friend int operator +=(int &i, Tile const &t);
    friend int operator +(int i, Tile const &t);
};
```

b) Overlagre operatorene **+** og **+=** for **`Tile`**-klassen slik at det blir enkelt å summere tallverdiene av en samling brikker. Eksempel på bruk av begge operatorene er vist under.

```
vector<Tile> tiles;
tiles.push_back(Tile('K', 3));
tiles.push_back(Tile('O', 3));
tiles.push_back(Tile('N', 1));
tiles.push_back(Tile('T', 1));
```

```
int sum1 = 0, sum2 = 0;
for (int i = 0; i < tiles.size(); i++){
   sum1 += tiles[i];
}
for (int i = 0; i < tiles.size(); i++){
   sum2 = sum2 + tiles[i];
}
```

```
Siden operatorene har int som venstre operand og Tile som høyre operand
kan de IKKE implementeres som medlemmer. For += MÅ venstre int-operand
være call-by-reference siden denne skal oppdateres. Om vi sender over
Tile som const referense er litt mindre viktig. For + operatoren er det
egentlig ingen grunn til at int skal være call-by-reference. Alle opera-
torer har returtyper og for += returnerer vi verdien etter oppdatering

int operator +=(int &i, Tile const &t){
    i += t.value;
    return i; //eller bare en linje: return i += t.value;
}
int operator +(int i, Tile const &t){
    return i + t.value;
}
```

Brettet som brukes i Scrabble og Wordfeud har 15x15 felt. Noen av feltene er bonusfelt av typen DL, TL, DW og TW og gir ekstra poeng. For å representere brettet og implementere deler av spillereglene skal vi lage en klasse kalt **`Board`**.

c) Vis eller forklar medlemsvariabler og konstruktører du trenger for klassen **`Board`**. Hvert felt skal kunne adresseres med x,y koordinater, det skal være mulig å "legge" en brikke på et felt, brettet må vite om et felt er et bonusfelt eller ikke, du må kunne sjekke om et felt er tomt eller om det allerede ligger en brikke der og eventuelt hvilken brikke som ligger på feltet. TIPS: Her bør du lage en egendefinert datatype for felt som f.eks. har en enum-variabel for felttypen og en peker til en brikke.

```
Medlemsvariabelen vi trenger er en todimensjonal tabell. Siden vi skal
lagre forskjellig informasjon for hvert felt, kan vi lage oss en egen
datatype for dette (struct eller class). Vi bruker peker til Tile for å
kunne bruke NULL for tomme felt. For å skille mellom forskjellige typer
bonusfelt lager vi en enum med konstanter for de forskjellige bonusfel-
tene og en konstant for NONE (for felt som ikke er bonusfelt).

enum FIELDTYPE {DL, TL, DW, TW, NONE};

class Field{
public:
  FIELDTYPE fieldtype;
  Tile *tile;
  int x,y;
public:
  Field(){ //inline konstruktør som setter defaultverdier
    fieldtype = NONE;
    tile = NULL;
    x = y = 0;
  }
};

Medlemsvariablene x og y er noe du trenger i de seinere deloppgavene (og
ikke noe som er nødvendig for denne deloppgaven).

class Board{
public:
    Field board[15][15];
public:
    Board();
};
Board::Board(){
  for (int x = 0; x < 15; x++){
    for (int y = 0; y < 15; y++){
      board[x][y].tile = NULL;
      board[x][y].x = x;
      board[x][y].y = y;
    }
  }
  //Her kunne vi kodet markering av bonusfelt
}
```

d) Implementer følgende medlemsfunksjoner i Board-klassen:

```
bool placeTile(Tile *t, int x, int y);
bool play();
```

Spilleren som har tur legger én og én brikke ved hjelp av ***placeTile****. Spilleren kan legge på et hvilket som helst felt hvor det ikke ligger noen brikke fra før. Når spilleren er ferdig med å legge brikker gir han beskjed til brettet med* ***play()****. Denne funksjonen tester om brikkene er lagt riktig ved hjelp av medlemsfunksjonen* ***bool isValidPlay()*** *- som du skal implementere i deloppgaven under. Hvis brikkene er lagt riktig returnerer play()* ***true****. Hvis ikke skal alle brikker som ble lagt i løpet av spilleren sin tur fjernes igjen fra brettet og play() skal returnere* ***false****. TIPS: Her kan det være greit at klassen "husker" hvilke felt en spiller legger brikker på, slik at det er enkelt å fjerne brikkene igjen fra brettet.*

```
I Board-klassen legger vi til en medlemsvariabel for å lagre pekere til
felt som en spiller har lagt på. Da blir det enkelt å fjerne brikker igjen
hvis de ikke er lagt i henhold til spillets regler.

vector<Field *> played //ny medlemsvariabel i Board

bool Board::placeTile(Tile *t, int x, int y){
    if(board[x][y].tile == NULL){
        board[x][y].tile = t;
        played.push_back(&(board[x][y]));
        return true;
    }else{
        return false;
    }
}

bool Board::play(){
  bool valid = false;
  if (isValid()){
    valid = true;
  }else{
    //"fjerner" brikkene fra brettet igjen
    for (int i = 0; i < played.size(); i++){
      played[i]->tile = NULL;
    }
    valid = false; //egentlig en unødvendig linje
  }
  played.clear();
  return valid;
}
```

e) Implementer funksjonen **bool Board::isValidPlay()** som ble brukt i oppgaven over. Den skal sjekke at spillets regler for plassering av brikker er fulgt. Funksjonen returnerer **true** hvis brikkene er lagt på lovlige plasser, og **false** hvis ikke. Du kan basere deg på skissen til løsning som er beskrevet på neste side. NB! Her er det viktig at du dekomponerer løsningen i mindre funksjoner og viser hvordan disse kombineres til en helhetlig løsning.

Testingen på om brikkene er lagt riktig gjøres etter at en spiller har lagt ferdig sine brikker på brettet. Hvis du har fulgt tipset i oppgave d) har du også en variabel som holder styr på hvilke felt en spiller har lagt på. Da kan du sjekke med brettet at det ligger en brikke i midten

og at alle brikker ligger inntil hverandre, og du kan bruke variabelen for "brikker som er lagt av spilleren" til å sjekke at alle brikker er på linje enten vertikalt eller horisontalt.

NB! Helt riktig logikk for å sjekke er litt underordnet, det viktigste er at du viser bruk av flere funksjoner som hver gjør sin del av jobben. Logikken som er beskrevet over er en litt forenklet løsning.

```
bool Board::isValid(){
  if (emptyBoard){
  //spesialtester som må gjøres for første spiller
    if (played.size() < 2){
      return false;
    }
    if (board[7][7].tile == NULL){
      return false;
    }
  }
  //tester som gjelder for alle spillere
  if (!(verticalTilesTest() || horizontalTilesTest())){
    return false;
  }
  if (!adjacentTilesTest()){
    return false;
  }
  if (!wordTest()){ //denne er bare tatt med for å vise hvor
    return false;   //deloppgave 3g skal inn
  }
  return true;
}
```

```
I tillegg bør hver enkelt testfunksjon være dekomponert i delfunksjoner
der det er naturlig. Her ser vi etter gode forsøk på implementasjoner og
at du viser generell programmeringskompetanse og problemforståelse. Vi
tar selvsagt hensyn til at det er svært begrenset med tid på eksamen til
å tenke ut og lage en løsning.
Se koden i siste del av LF for en mulig løsning(men husk at dette er en
oppgave som kan løses på mange måter).
```

f) Ord som legges på brettet skal være vanlige, kjente ord. For å kunne sjekke om et ord er "lovlig", kan vi bruke ei ordbok. Implementer klassen **`Dictionary`** til dette formålet med medlemsfunksjonen **`bool isValid(const string &word)`**. Konstuktøren skal ta inn filnavnet og lese fra fila inn i en egnet datatype. I ordbok-fila er det brukt linjeskift for å skille mellom ordene.

```
Her er en løsning vist med inline implementasjoner.
I oppgaven står det ingenting om hvordan du skal håndtere situasjoner
hvor fila ikke lar seg åpne/lese, og derfor tester vi bare om fila er
åpnet før vi leser. Husk å bruke close() når vi er ferdige med fila.
```

```
class Dictionary{
private:
    set<string> words; //mer effektiv å bruke set enn vector
public:
    Dictionary(const string &filename){
        ifstream infile(filename.c_str());
        if (infile){
            string w;
            while(getline(infile, w)){
                words.insert(w);
            }
            infile.close();
        }
    }
    bool isValid(const string &word){
        if ((words.find(word) != words.end())){
            return true;
        }
        return false;
    }
};
```

g) Når en spiller har lagt sine brikker må vi finne ut om ordene som er laget er lovlige. Lag en medlemsfunksjon som finner alle ordene som en bruker har laget og sjekker om de er lovlige.

```
Her er det også en god ide å lage delfunksjoner for løsningen. Her bruker
vi funksjoner som finner horisontale og vertikale ord gitt en Tile* som
input. Vi har også lagt til en medlemsvariabel dict i Boardklassen av
typen Dictionary. Her er det selvsagt mulig med mange løsningsstrategier.
bool Board::wordTest(){
  if (verticalTilesTest()){
    if (!dict.isValid(getVerticalWord(played[0])){
      return false;
    }
    for (int i = 0; i < played.size(); i++){
      string hs = getHorisontalWord(played[i]);
      if (hs.size() > 1 && !dict.isValid(hs)){
        return false;;
      }
    }
  }
  if (horizontalTilesTest()){
    if (!dict.isValid(getHorisontalWord(played[0])){
      return false;
    }
    for (int i = 0; i < played.size(); i++){
      string vs = getVerticalWord(played[i]);
      if (vs.size() > 1 && !dict.isValid(vs)){
        return false;
      }
    }
  }
  return true;
}
```

Se vedlegget for kode til fullstendig løsning.

h) Og hvis du har tid igjen: implementer en funksjon for å regne ut poengsummen en spiller skal ha etter en runde. Poengene beregnes først for hvert ord og deretter summeres disse. Bonusfelt som brukeren legger på skal medregnes. DL gir dobbelt bokstavpoeng og TL gir trippelt bokstavpoeng. DW gir dobbel ordsum og TW gir trippelt ordsum. Bokstavbonus regnes ut først og deretter ordbonusene. Hvis en bokstav brukes i to ord skal den telles med i begge inklusive eventuell bokstavbonus.

Dette blir litt samme logikk som for forrige oppgave. Her har vi laget egne funksjoner for å beregne score for horisontale og vertikale ord. Se kode på slutten av LF for en mer fullstendig løsning.

```
int Board::calculateScore(){
    int score = 0;
    if (verticalTilesTest()){
        score += calculateVerticalWordScore(played[0]);
        for (int i = 0; i < played.size(); i++){
            score += calculateHorizontalWordScore(played[i]);
        }
    }else if(horizontalTilesTest()){
        score += calculateHorizontalWordScore(played[0]);
        for (int i = 0; i < played.size(); i++){
            score += calculateVerticalWordScore(played[i]);
        }
    }
    return score;
}

int Board::calculateVerticalWordScore(Field *f){
  int score = 0;
  int x = f->x;
  //Finner første og siste felt i vertikalt ord
  int y1 = uppermost(f)->y;
  int y2 = lowermost(f)->y;
  if (y1 == y2){
    //ord med bare en bokstav er ikke noe ord
    return 0;
  }
  int wordbonus = 1;
  for (int y = y1; y <= y2; y++){
    if (find(played.begin(),played.end(),&board[x][y])!= played.end()){
    //brikken er lagt av denne spilleren og da skal vi regne bonusfelt
      if (board[x][y].fieldtype == NONE){
        score += board[x][y].tile->getValue();
      }else if (board[x][y].fieldtype == DL){
        score += 2 * board[x][y].tile->getValue();
      }else if(board[x][y].fieldtype == TL){
        score += 3 * board[x][y].tile->getValue();
      }else if (board[x][y].fieldtype == DW){
        wordbonus *= 2;
      }else if (board[x][y].fieldtype == TW){
```

```
          wordbonus *= 3;
        }
      }else{
        score += board[x][y].tile->getValue();
      }
    }
    score *= wordbonus;
    return score;
}
```

NB! Merk at løsningen som er laget ikke er representativ for hva som kreves av svar i deloppgave 3 e,f og g. Det er heller ikke noen garanti for at løsningsforslaget er feilfritt ;-)

Appendiks 1

Regler for Scrabble
(fritt etter de offisielle Scrabble-reglene og med forbehold om at det kan være feil)

Scrabble og Wordfeud spilles på et brett med 15x15 felt. En illustrasjon av det klassiske Scrabble-brettet finner du på siste side. Noen felt er såkalte bonusfelt av typen DL (double letter), TL (triple letter), DW (double word) og TW (triple word). Disse feltene gir ekstra poeng. Midten er markert med en stjerne siden det første ordet som legges må dekke midten. Utover dette har ikke stjerna noen spesiell funksjonalitet. Wordfeud er en variant av Scrabble tilpasset smartphones og pads. I Wordfeud kan du også velge et random-brett hvor bonusfeltene er tilfeldig plassert.

I Scrabble brukes 100 bokstavbrikker og de forskjellige bokstavene har forskjellig verdi. Eksempelvis har bokstavene A, N, R, S, T verdien 1 siden de er lette å bruke til å lage ord, mens D og G har verdien 2, K har verdien 5 osv. Distribusjon av bokstavbrikker og verdien til bokstavene kan varierer fra språk til språk. Wordfeud har sitt eget oppsett som avviker litt fra det klassiske spillet. Det finnes også to blanke brikker som kan brukes som hvilken som helst bokstav, men blanke brikker har ingen verdi (eller verdien 0).

Den første spilleren legger to eller flere av sine brikker slik at de former et ord enten bortover eller nedover. Diagonale ord er ikke tillatt og første ord må dekke midterste felt (det som er merket med stjerne). Alle brikkene som den første spilleren legger må selvsagt ligge inntil hverandre.

Alle ord som legges skal være gyldige ord. Når man spiller brettspillet må man avtale på forhånd hva som er lovlige ord og man kan f.eks. bruke en ordbok for å avgjøre hvis det er tvil. Spiller du Wordfeud på en smartphone eller pad vil programmet automatisk sjekke om ordene som legges er lovlige ved at det gjøres et oppslag i ei ordliste.

Neste spiller legger en eller flere bokstaver slik at de former et eller flere nye ord eller endrer på ord som allerede er lagt. Alle bokstaver som legges må plasseres på samme linje enten bortover eller nedover og alle brikker må ligge inntil en annen brikke. Spilleren kan legge til bokstaver før og etter brikker som allerede ligger på brettet, eller legge brikker parallelt med ord som allerede ligger på brettet. Slik kan det lages ett eller flere nye ord både horisontalt og vertikalt i samme runde. Se illustrasjon på slutten for eksempel på hvordan spillet kan utvikle seg. Den første spilleren legger ordet “SPANSK” slik at det dekker midtfeltet. Neste spiller legger bokstavene P, E og T slik at det former ordet “PENT” sammen med bokstaven forrige spiller la. Neste spiller legger bokstaven S og former ordene “SE” og “SA”. Siste spiller legger bokstavene I, E og T og får ordene “SEI”, “ISET” og “TE”.

Spillet har også regler om å melde pass, hvordan brikker skal trekkes, hvordan spillet skal avsluttes og mye mer, men de hopper vi over siden det ikke er relevant for oppgaven.

Poengene en spiller får etter sin tur er summen av bokstavene i alle ordene som er laget eller endret plus eventuelle ekstrapoeng hvis en eller flere brikker legges på et bonusfelt. DL gir 2x bokstavpoeng, TL gir 3x bokstavpoeng, DW gir 2x ordpoeng og TW gir 3x ordpoeng. Bokstavbonus regnes ut først. Bokstaver og bonusfelt som inngår i flere ord som en bruker lager, telles for hvert ord. Etter at det er lagt en brikke på et bonusfelt teller det ikke for andre spillere.

Spør faglærer som går eksamensrunden hvis du trenger en bedre forklaring på spillet. Oppgaven er ikke en øvelse i å forstå reglene og du skal få den hjelpen du trenger.

TW			DL				TW				DL			TW
	DW				TL				TL				DW	
		DW				DL		DL				DW		
DL			DW				DL				DW			
				DW						DW				
	TL				TL				TL				TL	
		DL				DL		DL				DL		
TW			DL				☆				DL			TW
		DL				DL		DL				DL		
	TL				TL				TL				TL	
				DW						DW				
DL			DW				DL				DW			
		DW				DL		DL				DW		
	DW												DW	
TW							TW							TW

1)

S	P	A	N	S	K

2)

			P		
			E		
S	P	A	N	S	K
			T		

3)

			P		
		S	E		
S	P	A	N	S	K
			T		

4)

			P		
		S	E	I	
S	P	A	N	S	K
			T	E	
				T	

Appendiks 1

Kode for en fullstendig implementasjon av oppgave 3

NB! Koden er ikke representativ for hva som kreves på eksamen!

```
#include <iostream>
#include <string>
#include <vector>
#include <set>
#include <algorithm>
#include <fstream>
using namespace std;

class Dictionary{
private:
    set<string> words;
public:
    Dictionary(const string &filename){
        ifstream infile(filename.c_str());
        if (infile){
            string w;
            while(getline(infile, w)){
                words.insert(w);
            }
            infile.close();
        }
    }
    bool isValid(const string &word){
        //if (words.find(word) != words.end()){
        //    return true;
        //}else{
        //    cout << word << " ikke godkjent" << endl;
        //}
        //return false;
        //Vi godkjenner alle ord mens vi prøver ut programmet
        return true;
    }
};

enum FIELDTYPE {DL, TL, DW, TW, NONE};

class Tile{
public:
    char letter;
    int value;
public:
    char getLetter() {return letter;}
    int getValue() {return value;}
    Tile(char letter, int value): letter(letter), value(value){}
    friend int operator +=(int &i, Tile const &t);
    friend int operator +(int i, Tile const &t);
};

int operator +=(int &i, Tile const &t){
    i += t.value;
    return i;
}

int operator +(int i, Tile const &t){
    return i + t.value;
}



class Field{
```

```
public:
    FIELDTYPE fieldtype;
    Tile *tile;
    int x,y;
public:
    Field(){
        fieldtype = NONE;
        tile = NULL;
        x = y = 0;
    }
    friend class Board;
    friend bool operator <(const Field&, const Field &);
};

class Board{
private:
    Field board[15][15];
    bool emptyBoard; //for å kunne teste om det er første ord som legges
    vector<Field *> played;
    Dictionary dict;
    bool isValid();
    bool horizontalTilesTest(); //tester om alle brikkene er lagt horisontalt
    bool verticalTilesTest();   //tester om alle brikkene er lagt vertikalt
    bool adjacentTilesTest();   //sjekker om brikker ligger etter hverandre
    Field* leftmost(Field* f);  //finner første felt for et horisontalt ord
    Field* rightmost(Field* f); //finner siste felt for et horisontalt ord
    Field* uppermost(Field* f); //finner første felt for et vertikalt ord
    Field* lowermost(Field* f); //finner siste felt for et vertikalt ord
    string getHorisontalWord(Field *f); //finner ordet som brikken
    string getVerticalWord(Field *f);   //på dette feltet er del av
    bool wordTest();  //tester på om ordene er gyldige
    int calculateScore();
    int calculateVerticalWordScore(Field *f);
    int calculateHorizontalWordScore(Field *f);
    void sortPlayed();  //sorterer feltene en spiller har lagt brikker på
public:
    Board();
    bool placeTile(Tile *t, int x, int y);
    bool play();
    friend ostream& operator<<(ostream& out, const Board &b);
};

int main(int argc, const char * argv[]){
    Board b;
    b.placeTile(new Tile('S', 1), 4, 7);
    b.placeTile(new Tile('P', 3), 5, 7);
    b.placeTile(new Tile('A', 1), 6, 7);
    b.placeTile(new Tile('N', 1), 7, 7);
    b.placeTile(new Tile('S', 1), 8, 7);
    b.placeTile(new Tile('K', 5), 9, 7);
    b.play();
    cout << b;

    b.placeTile(new Tile('P', 3), 7, 5);
    b.placeTile(new Tile('E', 1), 7, 6);
    b.placeTile(new Tile('T', 1), 7, 8);
    b.play();
    cout << b;

    b.placeTile(new Tile('S', 1), 6, 6);
    b.play();
    cout << b;

    b.placeTile(new Tile('I', 1), 8, 6);
    b.placeTile(new Tile('E', 1), 8, 8);
    b.placeTile(new Tile('T', 1), 8, 9);
    b.play();
    cout << b;
}
```

```
Board::Board():dict("dictionary.txt"){
    emptyBoard = true;
    for (int x = 0; x < 15; x++){
        for (int y = 0; y < 15; y++){
            board[x][y].x = x;
            board[x][y].y = y;
            board[x][y].tile = NULL;
        }
    }
    board[6][6].fieldtype = DL;
    board[6][8].fieldtype = DL;
    board[8][6].fieldtype = DL;
    board[8][8].fieldtype = DL;
}

bool Board::placeTile(Tile *t, int x, int y){
    if(board[x][y].tile == NULL){
        board[x][y].tile = t;
        played.push_back(&(board[x][y]));
        return true;
    }else{
        return false;
    }
}

bool Board::play(){
    bool valid = false;
    sortPlayed();
    cout << "Played: ";
    for (int i = 0; i < played.size(); i++){
        cout << played[i]->tile->getLetter();
    }
    cout << endl;
    if (isValid()){
        if (emptyBoard){
            emptyBoard = false;
        }
        valid = true;
    }else{
        //setter peker til tile til NULL for alle felt det er lagt på
        for (int i = 0; i < played.size(); i++){
            played[i]->tile = NULL;
        }
        valid = false;
    }
    if (valid){
        cout << "Score = " << calculateScore() << endl;
    }else{
        cout << "Ikke godkjent!" << endl;
    }
    played.clear();
    return valid;
}

bool Board::isValid(){
    sortPlayed();
    if (emptyBoard){
        //spesialtester som må gjøres for første spiller
        if (played.size() < 2){
            return false;
        }
        if (board[7][7].tile == NULL){
            return false;
        }
    }
    //tester som gjelder for alle spillere
    if (!(verticalTilesTest() || horizontalTilesTest())){
        return false;
    }
    if (!adjacentTilesTest()){
```

```
        return false;
    }
    if (!wordTest()){
        return false;
    }
    return true;
}


bool Board::horizontalTilesTest(){
    //alle brikker må være lagt i samme rad (ha samme verdi for y)
    int y = played[0]->y;
    for (int i = 1; i < played.size(); i++){
        if (played[i]->y != y){
            return false;
        }
    }
    return true;
}

bool Board::verticalTilesTest(){
    //alle brikker må være lagt i samme kolonne (ha samme verdi for x)
    int x = played[0]->x;
    for (int i = 1; i < played.size(); i++){
        if (played[i]->x != x){
            return false;
        }
    }
    return true;
}

Field* Board::leftmost(Field *f){
    //Denne kan vi bruke til å finne første felt i et horisontalt ord på brettet
    int x = f->x;
    int y = f->y;
    while (x > 0) {
        if (board[x-1][y].tile == NULL){
            return &board[x][y];
        }
        x--;
    }
    return &board[x][y];
}

Field* Board::rightmost(Field *f){
    //Denne kan vi bruke til å finne siste felt i et horisontalt ord på brettet
    int x = f->x;
    int y = f->y;
    while (x < 14) {
        if (board[x+1][y].tile == NULL){
            return &board[x][y];
        }
        x++;
    }
    return &board[x][y];
}

Field* Board::uppermost(Field *f){
    //Denne kan vi bruke til å finne første felt i et vertikalt ord på brettet
    //kanskje litt misvisende navn siden vi finner feltet i ordet med lavest y-verdi
    //fordi vi regner 0,0 som øvre, venstre hjørne i brettet
    int x = f->x;
    int y = f->y;
    while (y > 0) {
        if (board[x][y-1].tile == NULL){
            return &board[x][y];
        }
        y--;
    }
    return &board[x][y];
```

```
}

Field* Board::lowermost(Field *f){
    //Denne kan vi bruke til å finne siste felt i et vertikalt ord på brettet
    int x = f->x;
    int y = f->y;
    while (y < 14) {
        if (board[x][y+1].tile == NULL){
            return &board[x][y];
        }
        y++;
    }
    return &board[x][y];
}


bool Board::adjacentTilesTest(){
    if (emptyBoard){
        //for første ord holder det å sjekke at brikkene er lagt etter hverandre
        //siden vi har sortert feltene det er lagt på kan vi sjekke
        //at enten x eller y øker med 1 for hvert felt i played-vectoren
        if (verticalTilesTest()){
            for (int i = 0; i < played.size() - 1; i++){
                if ((played[i]->y + 1) != played[i+1]->y){
                    return false;
                }
            }
        }
        if (horizontalTilesTest()){
            for (int i = 0; i < played.size() - 1; i++){
                if ((played[i]->x + 1) != played[i+1]->x){
                    return false;
                }
            }
        }
        return true;
    }
    else{
        //for suksessive runder kan vi teste litt annerledes
        if (horizontalTilesTest()){
            //vi finner første felt i ordet som sist brikke i played er lagt på
            //og siste felt som første brikke i played er lagt på
            //hvis antallet felt er større enn antallet brikker som er lagt
            //betyr det at vi har lagt et ord som bygger videre på en eksisterene bokstav
            //eller et eksisterende ord
            //litt tricky logikk å forstå, men den er forhåpentligvis riktig
            Field *leftfield = leftmost(played[played.size() - 1]);
            Field *rightfield = rightmost(played[0]);
            if ((rightfield->x - leftfield->x) + 1 > played.size()){
                return true;
            }else if ((rightfield->x - leftfield->x) + 1 == played.size()){
                //I tilfellet ordet er lagt parallelt med et annet ord
                //kan vi sjekke at minst en av brikkene danner et ord vertikalt
                //vi iterer over alle feltene i played og sjekker om det finnes ord
                //vertikalt hvor første felt != siste felt
                for (int i = 0; i < played.size(); i++){
                    Field *upperfield = uppermost(played[i]);
                    Field *lowerfield = lowermost(played[i]);
                    if (upperfield != lowerfield){
                        return true;
                    }
                }
            }
        }else if (verticalTilesTest()){
            //samme logikk implementert for brikker som er lagt vertikalt
            Field *upperfield = uppermost(played[played.size() - 1]);
            Field *lowerfield = lowermost(played[0]);
            if ((lowerfield->y - upperfield->y) + 1 > played.size()){
                return true;
            }else if ((lowerfield->y - upperfield->y) + 1 == played.size()){
```

```
                for (int i = 0; i < played.size(); i++){
                    Field *leftfield = leftmost(played[i]);
                    Field *rightfield = rightmost(played[i]);
                    if (leftfield != rightfield){
                        return true;
                    }
                }
            }
        }
    }
    return false;
}


bool operator <(const Field& a, const Field & b){
    //vi overlagrer operator < slik at vi kan sortere de feltene en bruker har lagt på
    if (a.x < b.x){
        return true;
    }else if (a.x == b.x){
        return (a.y < b.y);
    }else{
        return false;
    }
}

ostream& operator<<(ostream& out, const Board &b){
    //for å undersøke og teste ut spillet er det greit
    //å kunne printe ut brettet
    for (int y = 0; y < 15; y++){
        for (int x = 0; x < 15; x++){
            if (b.board[x][y].tile != NULL){
                out << b.board[x][y].tile->letter << " ";
            }else{
                out << "*" << " ";
            }
        }
        out << endl;
    }
    out << endl;
    return out;
}

string Board::getVerticalWord(Field *f){
    Field *letter = uppermost(f);
    int x = letter->x;
    int y = letter->y;
    string s;
    while (y < 15 && board[x][y].tile != NULL){
        s += board[x][y++].tile->letter;
    }
    return s;
}

string Board::getHorisontalWord(Field *f){
    Field *letter = leftmost(f);
    int x = letter->x;
    int y = letter->y;
    string s;
    while (x < 15 && board[x][y].tile != NULL){
        s += board[x++][y].tile->letter;
    }
    return s;
}

bool Board::wordTest(){
    if (verticalTilesTest()){
        string vs = getVerticalWord(played[0]);
        if (!dict.isValid(vs)){
            return false;
        }
```

```
        for (int i = 0; i < played.size(); i++){
            string hs = getHorisontalWord(played[i]);
            if (hs.size() > 1 && !dict.isValid(hs)){
                return false;;
            }
        }
    }
    if (horizontalTilesTest()){
        string hs = getHorisontalWord(played[0]);
        if (!dict.isValid(hs)){
            return false;
        }
        for (int i = 0; i < played.size(); i++){
            string vs = getVerticalWord(played[i]);
            if (vs.size() > 1 && !dict.isValid(vs)){
                return false;
            }
        }
    }
    return true;
}

int Board::calculateVerticalWordScore(Field *f){
    //finner score for vertikalt ord
    int score = 0;
    int x = f->x;
    //Finner første og siste felt i vertikalt ord
    int y1 = uppermost(f)->y;
    int y2 = lowermost(f)->y;
    if (y1 == y2){
        //ord med bare en bokstav er ikke noe ord
        return 0;
    }
    int wordbonus = 1;
    for (int y = y1; y <= y2; y++){
        if (find(played.begin(), played.end(), &board[x][y]) != played.end()){
            //brikken er lagt av denne spilleren og da skal vi regne bonusfelt
            if (board[x][y].fieldtype == NONE){
                score += board[x][y].tile->getValue();
            }else if (board[x][y].fieldtype == DL){
                score += 2 * board[x][y].tile->getValue();
            }else if(board[x][y].fieldtype == TL){
                score += 3 * board[x][y].tile->getValue();
            }else if (board[x][y].fieldtype == DW){
                wordbonus *= 2;
            }else if (board[x][y].fieldtype == TW){
                wordbonus *= 3;
            }
        }else{
            score += board[x][y].tile->getValue();
        }
    }
    score *= wordbonus;
    return score;
}

int Board::calculateHorizontalWordScore(Field *f){
    //finner score for horisontalt ord
    int score = 0;
    int y = f->y;
    //Finner første og siste felt i horisontalt ord
    int x1 = leftmost(f)->x;
    int x2 = rightmost(f)->x;
    if (x1 == x2){
        //ord med bare en bokstav er ikke noe ord
        return 0;
    }
    int wordbonus = 1;
    for (int x = x1; x <= x2; x++){
        if (find(played.begin(), played.end(), &board[x][y]) != played.end()){
```

```
            //brikken er lagt av denne spilleren og da skal vi regne bonusfelt
            if (board[x][y].fieldtype == NONE){
                score += board[x][y].tile->getValue();
            }else if (board[x][y].fieldtype == DL){
                score += 2 * board[x][y].tile->getValue();
            }else if(board[x][y].fieldtype == TL){
                score += 3 * board[x][y].tile->getValue();
            }else if (board[x][y].fieldtype == DW){
                wordbonus *= 2;
            }else if (board[x][y].fieldtype == TW){
                wordbonus *= 3;
            }
        }else{
            //og til slutt et eksempem på bruk av operatoren fra 3b
            score += *board[x][y].tile;
        }
    }
    score *= wordbonus;
    return score;
}

int Board::calculateScore(){
    int score = 0;
    if (verticalTilesTest()){
        score += calculateVerticalWordScore(played[0]);
        for (int i = 0; i < played.size(); i++){
            score += calculateHorizontalWordScore(played[i]);
        }
    }else if(horizontalTilesTest()){
        score += calculateHorizontalWordScore(played[0]);
        for (int i = 0; i < played.size(); i++){
            score += calculateVerticalWordScore(played[i]);
        }
    }

    return score;
}



void Board::sortPlayed(){
    for(int i = played.size() - 1; i > 0; --i){
        bool swapped = false;
        for (int j = 0; j < i; ++j){
            if (*(played[j+1]) < *(played[j])){
                swap(played[j], played[j + 1]);
                swapped = true;
            }
        }
        if (!swapped){
            return;
        }
    }
}
```