NTNU
Institutt for datateknikk
og informasjonsvitenskap

Tentativt LF til Eksamensoppgave i TDT4102 - Prosedyre- og objektorientert programmering

Faglig kontakt under eksamen: Trond Aalberg

Tlf: 97631088

Eksamendato:	**16. august**
Eksamenstid:	**09-13**
Hjelpemiddelkode:	**C: Spesifiserte trykte og håndskrevne hjelpemidler tillatt.**
	Bestemt, enkel kalkulator tillatt.
	Walter Savitch, Absolute C++ eller
	Lyle Loudon, C++ Pocket Reference.
Målform/språk:	**Bokmål**
Antall sider:	**7**
Ingen vedlegg	

Kontrollert av

Dato Sign

Merk! Studenter finner sensur i Studentweb. Har du spørsmål om din sensur må du kontakte instituttet ditt. Eksamenskontoret vil ikke kunne svare på slike spørsmål.

Generell introduksjon

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

Når det står "*implementer*" eller "*lag*" skal du skrive en fungerende implementasjon: hvis det handler om en funksjon skal du skrive deklarasjonen med returtype og parametertype(r) og hele funksjons-kroppen.

Når det står "*deklarer*" er vi kun interessert i funksjons- eller klassedeklarasjonen. Typisk vil dette være deklarasjoner du vanligvis finner i header-filer.

Hvis det står "*forklar*" står du fritt i hvordan du svarer, men bruk enkle kodelinjer og/eller korte tekst-forklaringer og vær kort og presis.

Dersom du mener at opplysninger mangler i en oppgaveformulering, gjør kort rede for de antagelser og forutsetninger du finner nødvendig.

All kode skal være i C++. Det er ikke nødvendig å ha med include-statement eller vise hvordan koden skal lagres i filer.

Hoveddelene av eksamensoppgaven teller med den andelen som er angitt i prosent. Enkelte deloppgaver er vesentlig mer vanskelig/arbeidskrevende enn andre og vil derfor telle mer.

`LF er kun veiledende og kan inneholde feil.`

Oppgave 1: Implementering av spillet Othello (50%)

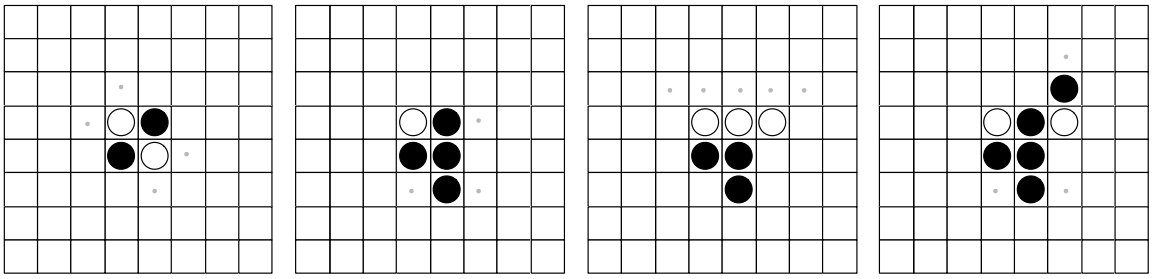
I denne oppgaven skal du implementere spillet "Othello". Dette er et klassisk spill for to hvor det brukes et brett med 8x8 felt og flate brikker som er hvite på den ene siden og svarte på den andre siden. Spillerne velger hver sin farge og spillet starter med 4 brikker lagt i midten (se figurene under).

For en beskrivelse av reglene har vi tatt med et utdrag fra lovene til Norges Othelloforbund:

§ 12 Svart gjør alltid første trekk, deretter trekker spillerne annenhver gang. Et trekk gjøres på følgende måte: Spilleren legger ned en brikke med egen farge opp, på et ledig felt i direkte forbindelse (diagonalt, vertikalt eller horisontalt) med et felt hvor det ligger en brikke fra før. Deretter vendes alle "innestengte" brikker. Som "innestengte" brikker defineres alle brikker som ligger på en rett linje mellom den brikken som nettopp ble lagt og nærmeste egne brikker, både diagonalt, vertikalt og horisontalt. Man kan kun gjøre et trekk når en eller flere av motstanderens brikker kan vendes. Kan man ikke vende noen av motstanderens brikker, mister man trekket og det er motstanderens tur igjen.

§ 14 Partiet er slutt når alle 64 felter er besatt, eller om ingen av spillerne kan gjøre noe trekk. Resultatet av partiet regnes i form av vinnerpoeng. Den spilleren som har flest brikker på brettet har vunnet og får 1 poeng i protokollen. Taperen får 0 poeng. Blir resultatet uavgjort får spillerne 1/2 poeng hver. I tillegg til vinnerpoeng skal også brikkeresultatet påtegnes i protokollen.

Figuren under viser hvordan spillet kan utvikle seg. Spillet starter med fire brikker lagt i midten som vist i første bilde. Feltene hvor svart kan legge er markert med prikk. I bilde nummer to har svart lagt en brikke i et felt og derfor fått vende den hvite brikken som var fanget mellom svarte brikker. Nå er det hvit sin tur og prikkene viser hvor hvit kan legge en brikke. I tredje bilde har hvit lagt en brikke og snudd en av svart sine brikker og slik fortsetter spillet.



I denne oppgaven skal du lage et program som lar en bruker spille Othello mot datamaskinen. Du skal definere en datatype som er egnet for å lagre spillets tilstand, funksjoner for å skrive ut spillets status og dialog med brukeren, samt logikken for å sjekke om et trekk er lovlig og hvilke brikker som skal vendes når en brikke er lagt. I deloppgavene skal du vise din kunnskap og erfaring i grunnleggende programmering i C++. Du bestemmer selv om du vil løse oppgaven objektorientert eller ikke. I evalueringen legges det vekt på at løsningen din er ryddig og lett å forstå for andre. Bruk gjerne kommentarer eller illustrer hvordan du mener koden er ment å fungere. Vi legger også vekt på at du har valgt en løsning som vil fungere i praksis, og at viser forståelse for hvordan de forskjellige delene av spillets logikk kan implementeres.

Deloppgave **`a-c`** er enkle oppgaver, mens deloppgave **`d`** er utfordrende. Oppgave **`e`** og **`f`** er middels vanskelige oppgaver og kan gjøres selv om du ikke har løst deloppgave **`d`**. Deloppgavene er med vilje laget som åpne spesifikasjoner hvor du selv må gjøre mange valg.

Vi forventer ikke en perfekt eller komplett løsning på alle deloppgaver, men ser etter fornuftige utkast som viser at du har forstått hva som skal gjøres og hvordan det kan løses.

a) Bestem en datatype for å representere et felt på brettet. Et felt kan være tomt eller ha en brikke som er hvit eller svart. Her er det mulig å definere en enum-type, men du kan også bruke andre løsninger.

```
//Som indikert i oppgavetekste kan det være greit å bruke en enum
enum Field {EMPTY, BLACK, WHITE};
```

b) Bruk et 8x8 array for brettet hvor feltene er av datatypen du definerte over og sørg for at brettet blir initialisert til riktig starttilstand.

```
//I LF er det brukt en klasse med en medlemsvariabel board
//Også greit å løse dette uten objektorientering
Field board[8][8];
//Konstruktøren skal sette alle felt til EMPTY og sette
//de midterste feltene til B/W iht bildet
//Vi setter alle felt til EMPTY ved å bruke board() i
//initialiseringslista, deretter setter vi spesifikke verdier i midten
Othello::Othello(): board(){
  board[3][4] = WHITE;
  board[4][3] = WHITE;
  board[3][3] = BLACK;
  board[4][4] = BLACK;
}
```

c) Lag funksjonalitet for å skrive ut brettets tilstand til skjerm.

```
//overlagrer operator<< for Field for å skrive ut et felt
ostream& operator<<(ostream& out, Field f){
  if (f == EMPTY){
    out << '*';
  }else if(f == BLACK){
      out << 'B';
  }else if(f == WHITE){
    out << 'W';
  }
  return out;
}
//overlagrer operator<< for å skrive ut brettet,
//deklarert som friend for å kunne adressere elementer direkte
ostream& operator<<(ostream &out, Othello &o){
//Skriver ut rad for rad, y er radindeks og x er kolonne
  for (int y = 0; y < Othello::SIZE; y++){
    for (int x = 0; x < Othello::SIZE; x++){
      out << o.board[x][y] << ' ';
    }
    out << endl;
  }
  return out;
}
```

d) Implementer funksjonalitet for å legge en brikke av en gitt farge på et angitt felt. Her er det sterkt anbefalt at du dekomponerer løsningen i flere hjelpefunksjoner siden det er mye som skal sjekkes og utføres. Funksjonen skal sjekke at et trekk er lovlig. I praksis bør du sjekke at brikken legges på et felt som er tomt, at brikken legges inntil en brikke av den motsatte

fargen, samt at trekket fanger minst en av motstanderens brikker i en eller flere retninger. Hvis trekket er lovlig skal brikken plasseres og du må finne motstanderens brikker som er fanget og vende disse.

Oppgaven kan løses med et moderat antall kodelinjer hvis du dekomponerer i fornuftige funksjoner og unngår duplisering av kode. Eksempelvis er det mulig å lage en generisk funksjon som kan sjekke i en hvilken som helst retning med parameter for delta x og delta y til å styre retningen du skal iterere i. Har du en slik funksjon er det enkelt å sjekke i alle retninger med ei løkke. Koden under kan du gi deg noen tips om hvordan dette kan løses:

```
for (int dx = -1; dx <= 1; dx++){
   for (int dy = -1; dy <= 1; dy++){
      if (dx == 0 && dy == 0){
         continue;
      }
      if (checkDirection(board, tile, x, y, dx, dy)){
         ....
         ....
      }
   }
```

```
//Returnerer true hvis trekket var lovlig
//Returnerer false hvis trekket ikke er lovlig og da gjøres det heller ingen endringer
bool Othello::put(int x, int y, Field f){
    //sjekker om feltet er tomt og at det er en lovlig indeks
    if (!inBounds(x, y) || board[x][y] != EMPTY){
        return false;
    }

    //siden vi sjekker i alle retninger må vi huske om en av retningene
    //er gyldig. Starter med false og endrer hvis vi finner retning hvor
    //vi fanger noen brikker
    bool valid = false;

    //løkke som sjekker i alle retninger
    for (int dx = -1; dx <= 1; dx++){
        for (int dy = -1; dy <= 1; dy++){
            if (dx == 0 && dy == 0){
                continue; // vi må hopper over denne kombinasjonen av dx og dy
            }
            if (checkDirection(f, x, y, dx, dy)){
                valid = true;
                    //Retningen er gyldig og derfor kan vi
                    //trygt snu brikkene i denne retningen
                int tempx = x;
                int tempy = y;
                while(board[tempx += dx][tempy += dy] != f){
                    board[tempx][tempy] = f;
                }
            }
        }
    }
    if (valid){
        //Plasser brikken
        board[x][y] = f;
    }
    return valid;
}
```

```
bool Othello::checkDirection(Field f, int x, int y, int dx, int dy){
    //Tester på første nabo
    if (!inBounds(x + dx, y + dy) ||
         (board[x+dx][y+dy] == EMPTY) ||  (board[x+dx][y+dy] == f)){
        return false;
    }
    //Vi leter videre i retningen etter en brikke av annen farge
    for (int nx = x + 2*dx, ny = y + 2*dy;; nx += dx, ny += dy){
        if (!inBounds(nx, ny)){
           return false; // vi har iterert utenfor brettet
        }
        if (board[nx][ny] == EMPTY){
           return false; // // vi har iterert til et tomt felt uten å finne noe
        }
        if (board[nx][ny] == f){
            return true;
        }
    }
}

bool Othello::inBounds(int x, int y){
    return (x >= 0) && (x < 8) && (y >= 0) && (y < 8);
}
```

e) Lag funksjonaliteten som trengs for at en spiller skal kunne gjennomføre et spill mot maskinen. Dvs. bestemme hvem som skal ha hvilken farge, be spilleren om hvilket felt som en brikke skal plasseres på - eller la brukeren si pass hvis han ikke kan legge, si fra om trekket er ulovlig, vise brettet etter at et trekk er gjort. Du kan implementere en helt enkel "dum" logikk for maskinen sin tur - eksempelvis bare iterere over alle felt og legge på første lovlige plass. Du trenger også en funksjon for å finne ut om spillet er over og finne ut hvem som vant.

Dette kan løses på mange måter og derfor gir vi ikke noe konkret forslag i LF. Viktig i svaret er at du viser bruk av cin/cout i en dialog for brukeren, bruk av løkke for å veksle mellom bruker sin tur og maskina sin tur, en enkel funksjon for maskina sin tur, en funksjon for å teste om spillet er over.

f) Implementer det som trengs for undo-funksjonalitet. I Othello er det ikke lov å gjøre om et trekk når en brikke først er lagt, men for en som skal lære seg å bli god kan det være nyttig å kunne gjøre om et trekk for dermed å kunne eksperimentere med alternative trekk. Dette kan løses ved at du lagrer alle trekk som er utført. Husk at et lovlig trekk medfører at en eller flere brikker vendes - du må med andre ord lagre informasjon om hvilken brikke som ble lagt på hvilket felt, samt informasjon om hvilke brikker som ble snudd. Her kan det være fornuftig å lage seg en egen datatype for denne informasjonen samt bruke en vector e.l. for å lagre trekkene. Når en bruker velger undo, kan du da bare lese siste element i vectoren og reversere de endringene som dette trekket innebar.

```
//datatype for å huske et felt
struct Position{
    unsigned int x;
    unsigned int y;
};

//datatype for å huske alle endringer som er gjort i et trekk
struct Move{
    Position placed;
    vector<Position> flipped;
};

//legger til en medlemsvariabel vector<Move> moves; i klassen og endrer
//putfunksjonen med en variabel av typen Move, legger til alle felt hvor
//en brikke blir vendt
//hvis det er en valid trekk så legger vi til i undo-vectoren

//Må også ha med en undo funksjon
void Othello::undo(){
    //"un-flipper" brikkene
    for (Position p: moves.back().flipped){
        if (board[p.x][p.y] == WHITE){
            board[p.x][p.y] = BLACK;
        }else if (board[p.x][p.y] == BLACK){
            board[p.x][p.y] = WHITE;
        }
    }
    //fjerner brikken som ble lagt
    board[moves.back().placed.x][moves.back().placed.y] = EMPTY;
    //fjerner siste move fra stakken
    moves.pop_back();
 }
```

Oppgave 2: 3-dimensjonale tabeller (50%)

I øvingsopplegget er du blitt godt kjent med dynamisk allokerte tabeller med 1 eller 2 dimensjoner. Dette er tabeller hvor størrelsen kan bestemmes i kjøretid fordi vi dynamisk allokerer det minnet som trengs. I denne oppgaven skal du implementere tilsvarende løsning for en 3-dimensjonal tabell. Vi kan kalle denne strukturen for en kube (eng: cube) siden den kan visualiseres som en kube med bredde (x-aksen), høyde (y-aksen) og dybde (z-aksen). Elementene (verdiene) i kuben kan vi da adresse med x-,y-,z-indeks.

I praksis er det forskjellige teknikker som kan brukes for tabeller hvor størrelsen skal bestemmes i kjøretid. I denne oppgaven skal du implementere noen alternative løsninger for dette som separate klasser og løse andre oppgaver relatert til denne problemstillingen.

Først skal du lage en klasse **`Cube_SingleArrImpl`** hvor du allokerer et sammenhengene array av størrelsen **`[width*height*depth]`** og implementerer get- og set-funksjoner hvor du bruker omregning fra x,y,z til en enkelt-indeks. Gitt en 3-dimensjonal tabell og indeksen (x,y,z) så kan du bruke følgende formel til å regne om til "en-dimensjonal" indeks: **`indeks = x + SIZEX * (y + z * SIZEY)`**, hvor **`SIZEX`** og **`SIZEY`** er størrelse til 3d-tabellen langs disse dimensjonene.

Et første utkast til klassen er vist under:

```
class Cube_SingleArrImpl{
private:
   int* cube;
   int height;
   int width;
   int depth;
public:
   Cube_SingleArrImpl(int width, int height, int depth);
   int get(int x, int y, int z);
   void set(int x, int y, int z, int value);
   int getHeight();
   int getWidth();
   int getDepth();
   void printSliceZ(int z);
   ~Cube_SingleArrImpl();
};
```

a) Implementer en konstruktør for klassen:
`Cube_SingleArrImpl::Cube_SingleArrImpl(int width,int height,int depth);`

Konstruktøren skal allokere et array av riktig størrelse og ellers ta seg av det som konstruktøren bør gjøre for denne klassen.
Vis bruk av konstruktørens initialiserings-liste!
Du trenger ikke tenke på initialisering av verdiene i arrayet.

```
Cube_SingleArrayImpl::Cube_SingleArrayImpl(int width, int height, int depth):
    width(width), height(height), depth(depth){
    cube = new int[width*height*depth];
}
```

b) Implementer get- og set-funksjonene for klassen:

```
int Cube_SingleArrImpl::get(int x, int y, int z);
void Cube_SingleArrImpl::set(int x, int y, int z, int value);
```

Her må du bruke formelen for omregning fra 3d-indeks til 1-d indeks som er forklart i innledningen til deloppgaven.

```
int Cube_SingleArrayImpl::get(int x, int y, int z){
    return cube[x + width * (y + z * height)];
}

void Cube_SingleArrayImpl::set(int x, int y, int z, int value){
    cube[x + width * (y + z * height)] = value;
}
```

c) Implementer en funksjon for å skrive ut (til cout) en "slice" av 3-tabellen:

```
void Cube_SingleArrImpl::printSliceZ(int z);
```

Se for deg at kuben din kan deles opp i en samling 2D-tabeller. For hver z-indeks kan du skrive ut en 2-dimensjonal tabell som i praksis vil være en "skive" (slice) av kuben.
Denne funksjonen skal ta inn en z-indeksverdi, iterere over x- og y-aksene og skrive ut den 2-dimensjonale tabellen som ligger lagret på denne z-indeksen.

```
void Cube_SingleArrayImpl::printSliceZ(int index){
    for (int y = 0; y < height; y++){
        for (int x = 0; x < width; x++){
            cout << setw(4) << get(x, y, index);
        }
        cout << endl;
    }
}
```

d) Implementer destruktøren for klassen:

```
SingleArrImpl::~Cube_SingleArrImpl();
```

```
Cube_SingleArrayImpl::~Cube_SingleArrayImpl(){
    delete [] cube;
}
```

e) Dette er en klasse hvor det er nødvendig med en destruktør og hvis det er behov for en destruktør er det som oftest også annen funksjonalitet som er nødvendig. Hva mener du det nødvendig å implementere for å få en mer profesjonell implementasjon av klassen? Hvis du mener det ikke er nødvendig med noe mer i klassen skal også svaret begrunnes.

Her er det kopi-konstuktør og tilordningsoperator vi mener.

I de neste deloppgavene skal du implementere klassen **`Cube_MultiArrImpl`**. Deklarasjonen av denne er identisk med **`Cube_SingleArrImpl`**, med unntak av medlemsvariabelen **`cube`** som for denne klassen skal implementeres som **`int*** cube`**. I denne klassen skal minnet til den 3D-dimensjonale tabellen allokeres "rad" for "rad". Hver x-indeks vil være en peker til en tabell av int* (y-rader), som igjen vil være pekere til int-arrays (z-rader).

f) Implementer konstruktøren for denne klassen:

`Cube_MultiArrImpl::Cube_MultiArrImpl(int width, int height, int depth);`

Allokeringen er litt intrikat og må løses med løkke(r), men prinsippet er det samme som vi har vist for 2D-tabeller i øvinger/forelesinger.

```
Cube_MultiArrayImpl(int width, int height, int depth):
 width(width), height(height), depth(depth){
    cube = new int**[width];
    for (int x = 0; x < width; x++){
        cube[x] = new int*[height];
        for (int y = 0; y < height; y++){
            cube[x][y] = new int[depth];
        }
    }
}
```

g) Implementer get- og set-funksjonene for denne klassen:

```
int Cube_MultiArrImpl::get(int x, int y, int z);
void Cube_MultiArrImpl::set(int x, int y, int z, int value);
```

Her er vi ute etter enklest mulig syntaks/løsning for å adressere enkeltementene.

```
int get(int x, int y, int z){
    return cube[x][y][z];
}

void set(int x, int y, int z, int value){
    cube[x][y][z] = value;
}
```

h) Implementer en funksjon for å skrive ut (til cout) en slice av 3D-tabellen:

`void Cube_MultiArrImpl::printSliceZ(int z);`

Denne skal fungere på samme måte som for Cube_SingleArrImpl.

Her kan vi bruke eksakt samme implementasjon som for Cube_SingleArrImpl.

i) Implementer destruktøren for klassen:

`Cube_MultiArrImpl::~Cube_MultiArrImpl();`

```
~Cube_MultiArrayImpl(){
    for (int x = 0; x < width; x++){
        for (int y = 0; y < height; y++){
            delete [] cube[x][y];
        }
        delete [] cube[x];
    }
    delete [] cube;
}
```

j) I konstruktøren for **`Cube_MultiArrImpl`** allokerer du minne som mange separate arrays (hvis du har løst oppgaven slik den skal løses). 3D-tabeller med store verdier for width, height og depth kan potensielt bruke svært mye minne og det vil være en fare for at du bruker opp alt minne som er til rådighet på maskinen. Hvis dette skjer vil operatoren **`new []`** kaste unntaket **`bad_alloc`**. Vis hvordan du kan fange opp dette unntaket, rydde opp i min-

nebruken og videresende unntaket slik at programmet kan fortsette uten minnelekasje (bruker minne som ikke frigjøres selv om det ikke lenger er i bruk).

```
Cube_MultiArrayImpl(int width, int height, int depth):
 width(width), height(height), depth(depth){
    try{
        //her bruker vi initialisering til å sette alle
       //pekere til nullptr
        cube = nullptr;
        cube = new int**[width]{nullptr};
        for (int x = 0; x < width; x++){
            cube[x] = new int*[height]{nullptr};
            for (int y = 0; y < height; y++){
                cube[x][y] = new int[depth]();
            }
        }
    }catch(bad_alloc &ba){
        if (cube != nullptr){
            for (int x = 0; x < width; x++){
                if (cube[x] != nullptr){
                    for (int y = 0; y < height; y++){
                        //trenger ikke teste på nullptr her
                        delete [] cube[x][y];
                    }
                    delete [] cube[x];
                }
            }
            delete [] cube;
        }
        throw ba; //sender unntaket videre
    }
}
```

Neste trinn i denne oppgaven er å lage en supertype for disse to klassene. Målet med denne oppgaven er at du skal vise bruk av arv og prinsipper relatert til arv.

k) Implementer en supertype kalt **`Cube`** som begge klassen du har jobbet med til nå kan arve fra. Det er et mål at mest mulig av medlemsvariablene og -funksjonene som du har laget for **`Cube_SingleArrImpl`** og **`Cube_MultiArrImpl`** skal deklareres og evt. implementeres i supertypen **`Cube`**.

Du trenger ikke endre implementasjonene eller deklarasjoner for **`Cube_SingleArrImpl`** *og* **`Cube_MultiArrImpl`**. *Syntaks for arv og riktig implementasjon av en subtype skal du likevel vise i siste deloppgave.*

I deklarasjonen og implementasjonen av supertypen bør du ha med følgende:

1) Hvilke medlemsvariabler som kan deklareres for supertype og deres synlighet.
2) En konstruktør for supertypen
3) Hvilke funksjoner som kan deklareres på supertypen.
4) Deklarering av funksjoner som virtual/pure virtual slik at vi får polymorf oppførsel.
5) Hvordan destruktøren til supertypen må deklareres slik at du får riktig oppførsel når objekter av subtypene slettes via delete på pekere av typen **`Cube*`**.

```
class Cube{
private:
    int width;
    int height;
    int depth;
public:
    Cube(int width, int height, int depth){
        this->width = width;
        this->height = height;
        this->depth = depth;
    }

    //pure virtual funksjoner
    virtual int get(int x, int y, int z) = 0;
    virtual void set(int x, int y, int z, int value) = 0;

    //disse funksjonene kan settes til virtual
    //men trenger ikke være virtual siden vi ikke
    //har behov for redefinerte funksjoner i subtypene
    virtual int getWidth(){return width;}
    virtual int getHeight(){return height;}
    virtual int getDepth(){return depth;}

    virtual void printZSlice(int index){
        for (int y = 0; y < height; y++){
            for (int x = 0; x < width; x++){
                cout << setw(4) << get(x, y, index);
            }
            cout << endl;
        }
    }

    //må være vritual så vi sikrer oss at subtypens destruktør
    //blir kalt
    virtual ~Cube(){};
};
```

I) Deklarer og implementer en subtype av **`Cube`** kalt **`Cube_VectorImpl`** hvor du viser i praksis hvordan du arver fra supertypen og hva som må implementeres i subtypen. I denne subtypen skal du bruke en nøstet struktur av **`std::vector`** for å implementere en 3-dimensjonal tabell. Med nøstet mener vi en vector som igjen inneholder en vector osv. Siden en vector i utgangspunktet er tom kan du bruke **`vector<T>::resize(int)`** for å sette størrelsen på dimensjonene.

```
class Cube_VectorImpl: public Cube{
private:
    vector<vector<vector<int>>> cube;
public:
    Cube_VectorImpl(int width, int height, int depth):
     Cube(width, height, depth){
        cube.resize(width);
        for (int i = 0; i < width; i++){
            cube[i].resize(height);
            for (int j = 0; j < height; j++){
                cube[i][j].resize(depth);
            }
        }
    }

    int get(int x, int y, int z){
        return cube[x][y][z];
    }

    void set(int x, int y, int z, int value){
        cube[x][y][z] = value;
    }
};
```