NTNU
Institutt for datateknikk
og informasjonsvitenskap

Eksamensoppgave i
TDT4102 - Prosedyre- og objektorientert programmering

Faglig kontakt under eksamen: Trond Aalberg

Tlf: 97631088

Eksamendato:	**27. mai**
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:	**9**
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.

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 funksjoner som utgjør deler i et program, eller forskjellige deler av en eller flere klasser. Du kan velge selv om du vil løse dette trinnvis, eller om du vil lage en samlet implementasjon, men sørg for at det går tydelig frem hvilke spørsmål du har svart på hvor i koden din. Husk at funksjonene du lager i en deloppgave ofte er ment å skulle brukes i andre deloppgaver.

All kode skal være i C++. Det er ikke viktig å huske helt korrekt syntaks for bibliotekfunksjoner. Oppgaven krever ikke kjennskap til andre klasser og funksjoner enn de du har blitt godt kjent med i øvingsopplegget. Det er ikke nødvendig å ha med include-statement eller vise hvordan koden skal lagres i filer.

Hele oppgavesettet er arbeidskrevende og det er ikke forventet at alle skal klare alt. Tenk strategisk i forhold til ditt nivå og dine ambisjoner! Husk at tid du bruker på å lete i boka gir deg mindre tid til å svare på oppgaver. Deloppgavene i de "tematiske" oppgavene er organisert i en logisk rekkefølge, men det betyr IKKE at det er direkte sammenheng mellom vanskelighetsgrad og nummereringen av deloppgavene.

Hoveddelene av eksamensoppgaven 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.

Oppgave 1: Kodeforståelse (20%)

a) Hva skrives ut?

```
int x = 1;
int y = 10 / x--;
int z = 10 / --x;
cout << x << endl;
cout << y << endl;
cout << z << endl;
```

b) Hva skrives ut?

```
bool x = true;
bool y = false;
bool z = ( x || y ) && ! ( x && y );
cout << boolalpha << z << endl;
//boolapha gir utskrift av boolske
//verdier som teksten "true"/"false"
```

c) Hva skrives ut?

```
int x = 10;
int y = 4;
double z = x / y;
cout << z << endl;
```

d) Gitt følgende implementasjon av `func()`, hva skrives ut av funksjonskallet `func(12345);` ?

```
void func(int i){
    if (i >= 10){
        cout << i % 10;
        func(i / 10);
    }else{
        cout << i << endl;
    }
}
```

e) Følgende kode fungerer ikke som den skal. Finn feilen og forklar hva som må rettes.

```
int x = 5;
int y = 6;
if (x =! y){
  cout << x << " is different from " << y << endl;
}else{
  cout << x << " is not different from " << y << endl;
}
```

Koden kompilerer og kjører, men utskriften blir "0 is not different from 6". Du sjekker meldingene som kompilatoren gir og finner en "Warning" som sier "Using the result of an assignment as condition without paranthesis". Dette er en feilmelding fra XCode som indirekte gir en pekepinn om hva og hvor feilen er.

.

f) Hvor mange objekter av typen `Foo` instansieres i løpet av ett kall til funksjonen under?

```
void func(){
  Foo a;
  for (int i = 0; i < 10; i++){
    Foo *c = new Foo();
    a = *c;
  }
}
```

g) Hvor mange ganger kalles destruktøren til `Foo` i løpet av et kall til funksjonen over?

h) Skriv nye deklarasjoner for de av operatorene under der du MÅ bruke call-by-reference for å få korrekt oppførsel. Behold call-by-value der dette vil fungere.
Vi er bare interessert i riktige parametre og ser ikke på retur-typen.

```
Foo operator+(Foo lhs, Foo rhs);
Foo operator++(Foo rhs);//prefiks
Foo operator+=(Foo lhs, Foo rhs);
bool operator<(Foo lhs, Foo rhs);
```

Oppgave 2: Funksjoner (35%)

I denne oppgaven skal du implementere spillet "fire på rad". Dette er et klassisk brettspill som består av en ramme med 7 kolonner hvor du slipper ned brikker. Det er plass til 6 brikker i hver kolonne. To spillere med hver sin farge slipper ned en brikke om gangen og den som først får fire på rad - enten horisontalt, vertikalt eller på skrå - vinner. Brikker (eng. tiles) som slippes ned i samme kolonne vil selvsagt havne over hverandre.

I vår versjon av spillet skal vi bruke en todimensjonal array (tabell) for å representere spillbrettet og vi skal bruke en **`enum`** for å representere gyldige verdier i arrayet (**`EMPTY`**, **`RED`**, **`YELLOW`**). Vi regner **`[0][0]`** for å være øvre venstre hjørne i brettet. Vi lar første indeks i arrayet representere rad, og den første brikken som slippes ned i kolonnen helt til venstre vil havne på posisjon **`[5][0]`**. Den neste som legges i samme kolonne vil havne i **`[4][0]`**.

For å forenkle programmeringen har vi laget globale konstanter for størrelsen til brettet og vi har laget et typealias for array-typen slik at det blir lettere å skrive funksjonsparameter. Variabel- og parameterdeklarasjoner kan nå skrives som **`Board board`** i stedet for **`Tile board[ROWS][COLS]`**.

I **`main`**-funksjonen under har vi deklarert og initialisert en variabel for brettet kalt **`board`**. Vi benytter oss av default initialisering for å sette alle elementene til **`EMPTY`**.

I koden under har vi også tatt med en eksempelfunksjon som tester for om en gitt kolonne i brettet er full.

```
enum Tile {EMPTY = 0, RED, YELLOW};

//The board has a static size
const int ROWS = 6;
const int COLS = 7;

//convenient alias typename
using Board = Tile[ROWS][COLS];

//example function
bool isFull(Board board, int col){
   return board[0][col] != EMPTY;
}

int main(){
    Board board{};
}
```

a) Implementer **`ostream& operator<<(ostream& o, Tile t);`** slik at du kan skrive ut på skjerm en variabel av **`Tile t;`** ved å skrive **`cout << t;`**
Hvis verdien av t er EMPTY skal du skrive ut '', hvis verdien er RED skal du skrive ut 'R', og hvis verdien er YELLOW skal du skrive ut 'Y'.*

b) Implementer **`operator<<`** slik du kan skrive ut på skjerm en variabel **`Board b;`** ved å skrive **`cout << b;`**

Utskriften skal være som vist til høyre.

```
* * * * * * *
* * * * * * *
* * * * * * *
* * R R R R *
* * Y R Y Y R
* Y R R Y Y Y
```

c) Implementer funksjonen `int findLowestPlace(Board board, int column);`
Funksjonen skal finne og returnere den nederste ruten i en kolonne hvor en brikke vil havne hvis den slippes ned. Hvis det ikke er ledige plasser i en kolonne returnerer funksjonen -1. Tips: brettet er en vanlig to-dimensjonal tabell og du skal lete deg frem til "nederste" plass i en kolonne som er EMPTY. Husk at det vi konseptuelt tenker på som nederste rad er raden med høyest indeks.

d) Implementer en funksjon for å slippe ned en brikke i en kolonne:
`int put(Board board, int column, Tile tile);`
Brikken tile skal havne på nederste ledige plass i den angitte kolonnen. Du kan anta at `tile` *er en lovlig brikke (*`tile != EMPTY`*). Funksjonen skal returnere raden hvor brikken havnet. Hvis kolonnen allerede er full skal det ikke plasseres noen brikke og funksjonen skal returnere -1. Her skal du selvsagt gjøre bruk av funksjonen i c).*

e) Implementer en funksjon for å teste om en rad- og kolonne-indeks er gyldig:
`bool isOutOfBounds(int row, int column);`

f) Hva betyr datatypen `unsigned int`, og hvilke fordeler ville det vært å systematisk bruke denne datatypen for `row` og `column` i dette programmet?

g) Implementer en funksjon for å telle brikker av samme farge som er etter hverandre ut fra en gitt rute. Funksjonen starter med brikken i ruten `[row][column]` og leter i retningen angitt med `dr` og `dc` (hhv. delta row og delta column). Funksjonen skal returnerer antallet brikker inkludert startbrikken.
`int countTiles(Board board, int row, int column, int dr, int dc);`
Parametrene dr og dc angir retning for iterasjon over rutene i brettet. Vi bruker verdiene til å inkrementere rad og kolonneindeksene i iterasjonen funksjonen må gjøre og da vil eksempelvis dr = 0 og dc = 1 gi iterasjon horisontalt mot høyre side. Radindeksen holdes konstant, mens kolonneindeksen inkrementeres med 1 for hver iterasjon. Bruker vi dr = 0 og dc = -1 iterere vi horisontalt mot venstre.

h) Implementer en funksjon som tester på om det er fire på rad i horisontalt, vertikalt eller på skrå, med utgangspunkt i brikken i ruten `[row][column].`
`bool fourInARow(Board board, int row, int column);`
I svaret ditt holder det at du viser kode for en av retningene horisontalt, vertikalt eller på skrå. *Husk at en brikke kan plasseres mellom andre brikker. I eksempelet som er vist i oppgave b) kan du se for deg at siste brikke er lagt i kolonne 3 og falt ned på rad 3. Da skal vi teste alle retninger med utgangspunkt i [3][3]. I tillegg må vi teste begge veier. For å teste korrekt horisontalt må vi først sjekke til høyre med countTiles(board, 3, 3, 1, 0), deretter til venstre med countTiles(board, 3, 3, -1, 0).*

i) Anta at en spiller har valgt farge. Implementer en funksjon `playerTurn()` som skriver ut brettet, ber spiller om hvilken kolonne han vil legge en brikke i, plasserer brikken, tester for fire i rekke og returnerer `true` hvis spilleren vant denne runden.

j) Implementer en funksjon `machineTurn()` slik at en bruker kan spille mot maskina. Implementer en "dum maskin" som velger tilfeldig kolonne hvor det er ledig plass og ellers oppfører seg som bruker-funksjonen over.

k) Lag en `main`-funksjon og eventuelle hjelpefunksjoner du trenger for at en bruker skal kunne spille mot maskina. Programmet skal håndtere alle utfall av spillet og avslutte ryddig når spillet er over.

Oppgave 3: Minnehåndtering (25%)

Klasser som bruker dynamisk allokert minne må vanligvis implementere en destruktør og som oftest også egendefinert kopi-konstruktør og tilordningsoperator. Når vi lager slike medlemmer må vi bestemme hva slags kopiering vi vil implementere. I prinsippet kan vi velge mellom to løsninger. Klassen kan implementere "verdi-lignende" oppførsel eller klassen kan implementere "referanse-lignende" oppførsel.

- For klasser som har "verdi-lignende" oppførsel vil alle objekter ha eget dynamisk allokert minne som frigis av destruktøren. Når vi kopierer verdi-lignende objekter vil kopi og original være helt uavhengige. Endringer i kopien har ingen effekt på originalen og vice versa. Denne løsningen er du blitt godt kjent med i øvingsopplegget (omtalt som deep copy).
- For klasser som har "referanse-lignende" oppførsel vil flere objekter dele samme dynamisk allokerte minne. Når vi kopierer slike objekter vil kopi og original peke til samme data. Endringer som gjøres på kopien vil også gjelde for originalen og vice versa. Denne løsningen har vi ikke brukt i øvingsopplegget, men kalles for shallow copy.

I denne oppgaven skal du implementere begge disse løsningene som henholdsvis klassen **`ValArr`** og klassen **`RefArr`**. Begge klassene har en medlemsvariabel **`data`** som skal peke til et dynamisk array av **`int`.** Størrelsen på det dynamisk arrayet bestemmes av konstruktør-argumentet.

For den første klassen **`ValArr`** skal det ved kopiering allokeres et nytt array og verdiene skal kopieres fra originalen. Siden alle objekter av typen **`ValArr`** vil ha sitt eget allokerte minne skal dette frigis når objektet slettes.

For den andre klassen **`RefArr`** skal kopiering gi objekter som peker til samme dynamisk allokerte minne. Vi bruker en egen variabel **`use`** for å holde telling med hvor mange objekter som "bruker" samme minne. Variabelen **`use`** inkrementeres og dekrementeres etter hvert som kopier lages eller slettes og hvis **`use`** blir 0 skal de allokerte variablene frigis.

```
class ValArr{
private:
   int *data;
   int size;
public:
   ValArr(int size);
   ValArr(const ValArr& org);
   ValArr& operator=(ValArr rhs);
   int& operator[](int i);
   int getSize();
   ~ValArr();
};


int& ValArr::operator[](int i){
   return data[i];
}
int ValArr::getSize(){
   return size;
}
```

```
class RefArr{
private:
   int *data;
   int *size;
   int *use;
public:
   RefArr(int size);
   RefArr(const RefArr& org);
   RefArr& operator=(const RefArr& rhs);
   int& operator[](int i);
   int getSize();
   ~RefArr();
};


int& RefArr::operator[](int i){
   return data[i];
}
int RefArr::getSize(){
   return *size;
}
```

Oppgaven fortsetter på neste side!

a) Implementer konstruktøren **`ValArr::ValArr(int size)`**. Konstruktøren skal allokere minne og sørge for at **`data`** peker til et array av størrelsen **`size`**. Bruk initialiseringsliste.

b) Implementer kopi-konstruktøren for klassen **`ValArr`**, denne skal opprette et array av samme størrelse som argumentet **`org`** og kopiere alle verdiene.
NB! Denne skal du implementere uten bruk av ValArr::operator= .

c) Implementer destruktøren for klassen **`ValArr`**.

d) I koden under har vi implementert tilordningsoperatoren for **`ValArr`** med bruk av en teknnikk som kalles "copy-swap". Forklar hvordan denne enkle koden tar seg av alt som er nødvendig å gjøre ved tilordning (med destruktør og kopi-konstruktøren impl. som i b og c).

```
ValArr& ValArr::operator=(ValArr rhs){
  swap(data, rhs.data);
  swap(size, rhs.size);
  return *this;
}
```

e) Implementer konstruktøren **`RefArr::RefArr(int size)`**. Konstruktøren skal allokere minne og sørge for at **`data`** peker til et array av størrelsen **`size`**. Bruk initialiseringsliste. I denne klassen bruker vi en variabel **`use`** som settes til 1 av konstruktøren.

f) Implementer kopi-konstruktøren for klassen **`RefArr`**. Kopi-konstruktøren skal lage en kopi av argument-objektet som deler samme minne som originalen. Kopi-konstruktøren skal inkrementere variabelen **`use`** siden det nå er et nytt objekt som bruker det samme minnet.

g) Implementer destruktøren for klassen **`RefArr`**. Denne skal dekrementere **`use`** og hvis **`use`** blir lik 0 skal alle dynamisk allokerte medlemsvariabler frigis.

h) Implementer tilordningsoperatoren for klassen **`RefArr`** slik du mener den bør fungere og kommenter kort hvordan din implementasjon virker.

i) I **`RefArr`** har vi brukt pekere for både **`size`** og **`use`**. Må begge disse være pekere? Gi en kort forklaring.

j) Gjør om en av klassene til en template-klasse. Du trenger kun å vise klasse-deklarasjonen.

Oppgave 4: Arv og litt mer (20%)

Denne oppgaven er inspirert av typiske dataspill hvor forskjellige skapninger tar livet av hverandre med sinnrike angrep. I implementasjonen av slike spill kan vi gjøre utstrakt bruk av arv og lage subtyper med forskjellig egenskaper og oppførsel.

I koden under har vi vist base-klassen **`Creature`** . Medlemsvariabelen **`name`** brukes for å navngi skapningen. Variabelen **`strength`** står for skapningens styrke, **`hitpoints`** er en variabel for hvor mye liv en skapning har og **`defence`** er en variabel som brukes for å moderere innkommende skade. For å utføre en kamp mellom to skapninger kan vi bruke de to medlemsfunksjonene **`doDmg()`** og t**`akeDmg()`**. Funksjonen **`doDmg()`** brukes for å regne ut hvor mye skade et angrep utført av skapningen kan gjøre. Basisskaden alle skapninger gjør er et tilfeldig tall mellom 1 og **`strength`**. Funksjonen **`takeDmg()`** brukes for å påføre en skapning skade (noe som reduserer skapningens **`hitpoints`**). Når en skapning ikke har flere **`hitpoints`** igjen er skapningen død og **`isAlive()`** returnerer **`false`**.

Vi skal bare jobbe med noen subtyper, men du kan se for deg at spillet skal ha mange forskjellige subtyper med forskjellige egenskaper.

```
class Creature{
private:
     string name;
     int hitpoints;
     int strength;
     int defence;
protected:
     virtual int doDmg(){return rand() % strength + 1;}
     virtual void takeDmg(int dmg){hitpoints -= max(dmg - defence, 0);}
public:
     Creature(string name): name(name), strength(100),
                            hitpoints(1000), defence(10){}
     bool isAlive(){return hitpoints > 0;}
     void attack(Creature& c){c.takeDmg(this->doDmg());}
};
```

a) Deklarer og implementer en subtype **`Demon`** som arver fra **`Creature`** og en subtype **`Balrog`** som arver fra **`Demon`**. For **`Demon`** skal **`doDmg()`** returnere en verdi (skade) hvor du legger til 10% til det som beregnes i **`Creature`**-klassen.
For **`Balrog`** skal **`doDmg()`** returnere skaden som beregnes for Demon-klassen, men det er 10% sjanse for dobbel skade (**`if (rand() % 10 == 0){}`**).

b) Vis hvordan konstruktørene for **`Demon`** og **`Balrog`** må implementeres for at du skal kunne instansiere objekter f.eks. med **`Balrog b = Balrog("Bally")`**;

c) Hva betyr det at **`doDmg()`** og **`takeDmg()`** er **`protected`** og hvilken grunn har vi for å bruke denne modifikatoren?

d) Hvorfor har vi deklarert **`doDmg()`** og **`takeDmg()`** som **`virtual`**.

e) Se for deg at du har laget mange subtyper av **`Creature`** som alle har forskjellige egenskaper. Du lager en funksjon **`void duel(Creature c1, Creature c2)`** for å ha kamper mellom ulike typer. Dette virker ikke helt etter planen da alle objekter oppfører seg likt inne i funksjonen. Hva er feil med funksjonen og hvorfor fungerer den feil?

Du skal lage en løsning for å lese en eller flere skapninger fra fil. Alle skapninger skal instansieres med default verdier for **`hitpoints`**, **`strength`** og **`defence`**.

f) Bestem et format for å beskrive en eller flere skapninger i en tekstfil. I praksis trenger vi bare informasjon om skapningens type og navn.

g) Implementer en funksjon som leser inn en fil i formatet du bestemte over - og oppretter en samling **`Creature`**-objekter slik at skapningenes type og navn blir som listet i filen.

h) Lesing fra fil kan av og til feile: filnavn kan være feil eller det kan være andre grunner til at fila ikke kan leses. I vår innlesing er det også andre ting som kan gå galt, eksempelvis at dataene i fila ikke stemmer med formatet vi forventer etc. Vis hvordan du kan bruke unntaksmekanismen for å **`a)`** si fra at feil er oppstått og **`b)`** fange opp unntak på dertil egnet sted i koden. Du kan godt bruke bibliotekstypen **`exception`** i denne oppgaven. Den har en konstruktør som aksepterer en **`string`** du kan bruke for å lagre tekstinformasjon om feilen - og en funksjon **`string exception::what()`** som returnerer teksten som ble satt med konstruktøren.