**BOKMÅL**



**NTNU**
**Norges teknisk-naturvitenskapelige universitet**

**Fakultet for informasjonsteknologi, matematikk og elektroteknikk**

**Institutt for datateknikk og informasjonsvitenskap**

# TENTATIVT LØSNINGSFORSLAG
# KONTERINGSEKSAMEN I FAG
# TDT4102 Prosedyre og objektorientert programmering

## Onsdag 6. august 2008
## Kl. 09.00 – 13.00

## NB! Det er ofte mange måter å løse en oppgave på!

**Faglig kontakt under eksamen:**
Trond Aalberg, tlf (735) 9 79 52 / 976 31 088

**Tillatte hjelpemidler:**

- Absolute C++, Savitch
- C++ Pocket Reference

**Sensurdato:**
Sensuren faller 3 uker etter eksamen og gjøres deretter kjent på http://studweb.ntnu.no/.

**Prosentsatser viser hvor mye hver oppgave teller innen settet.**

**Merk: Alle programmeringsoppgaver skal besvares med kode i C++.**

# Lykke til!

## Generelt for alle oppgaver

- Merk at ikke alle deloppgaver krever at de foregående er løst, så ikke hopp over de resterende deloppgavene om én blir for vanskelig.
- Der det spesifikt står definer eller deklarer er vi kun interessert i funksjondeklarasjonen eller klassedeklarasjoner. Der det står implementer skal du vise implementasjon av funksjoner.
- Der det spesifikt spørres etter forklaringer og beskrivelser er det et av kravene at dette skal være med.
- NB! Det er ikke et krav at alle funksjoner/klasser skal være komplette, implementer kun det som er nødvendig for å besvare en oppgave!

### OPPGAVE 1 (30 %): Funksjoner og operatorer

a) Implementer funksjonen **`int findfirst(int a[], int size, int value)`** som leter gjennom en tabell (array) av heltall og returnerer indeksen til første forekomst av **`value`**. Hvis **`value`** ikke finnes i indeksen skal du returnere -1. Parameter **`size`** angir størrelsen på tabellen.

```
int findfirst(int a[], int size, int value){
   for (int i = 0; i < size; i ++){
      if (a[i] == value)
         return i;
   }
   return -1;
};

// Her ser vi etter riktig bruk av løkke
// return av posisjonen når verdien er funnet vil føre til at
// funkjsonen avsluttes når (og hvis) verdien er funnet
// kan også løses ved at man mellomlagrer posisjonen og
// returnerer etter å gått igjennom hele løkken,
// men da man man sørge for å kun mellomlagre første treff
```

b) Implementer en funksjon med navn **`rightShift`** som du kan bruke til å endre verdiene til tre heltallsvariabler (**`int`**). Etter funksjonskallet **`rightShift(a, b, c)`** skal variabel **`b`** ha fått verdien til **`a`**, variabel **`c`** skal ha fått verdien til **`b`** og variabel **`a`** skal ha verdien til **`c`**. Funksjonen skal ikke returnere noe (**`void`**).

```
void rightShift(int& a, int& b, int& c){
   int temp = c;
   c = b;
   b = a;
   a = temp;
};
// Her er det viktig å bruke referanseparameter og
// implementere med en temp-variabel og
// riktig urført rekkefølge av tilordning
```

Implementer funksjonen **`bool XOR(bool a, bool b)`** som returnerer true når den ene parameteren er **`true`** og den andre er **`false`**. NB! I implementasjonen får du kun lov til å benytte de logiske operatorene AND (**`&&`**), OR (**`||`**) og NOT (**`!`**) – ingen andre operatorer

```
XOR(true, true) -> false
XOR(false, false) -> false
XOR(true, false) -> true
XOR(false, true) -> true
```

er tillatt. Funksjonen skal gi følgende resultat:

```
bool XOR(bool a, bool b) {
   return (a || b) && !(a &&b);
};
// Her ser vi etter riktig bruk av operatorer og kompakt
løsning
// Husk at oppgaven spesifiserer at du kun skal bruke ||, &&
og !
```

c) Implementer en funksjon **`void divideByTwo(int x)`** som ved hjelp av rekursjon skriver ut tallrekken (inklusive tallet selv) du får ved gjentatte deleoperasjoner med 2. Tallene skal skrives ut med det minste tallet først. Hvis funksjonen kalles med tallet 64 skal den skrive ut tallrekken: 1, 2, 4, 8, 16, 32, 64. Hvis funksjonen kalles med tallet 60 skal den skrive ut 15, 30, 60 osv.

```
void divideByTwo(int x){
   if ((x % 2) == 0){
      divideByTwo(x/2);
   }
   cout << x << endl;
};
// Oppgaven MÅ løses med rekursjon
// For å få utskrift i riktig rekkefølge minst -> størst
// må du skrive ut ETTER rekursivt kall
```

d) I funksjons-headeren i koden under er det benyttet referanse-parameter. Forklar hva dette er og vis hvilke endringer som du må gjøre i swap hvis du i stedet hadde benyttet peker-parameter i swap-dekarasjonen.

```
void swap(int& x, int& y){
    int temp = y;
    y = x;
    x = temp;
}


int main(){
    int x = 1;
    int y = 2;
    swap (x, y);
    //nå skal x ha verdien 2 og y ha verdien 1
}

// Hva må du endre i swap hvis header i stedet
// hadde vært swap(int* x, int* y) og
// funksjonskallet hadde vært swap(&x, &y)?
```

```
void mySwap(int *a, int *b){
    int temp = *a;
    *a = *b;
    *b = temp;
}
// Poenget i denne oppgaven er å vise bruk av
// derefereringsoperatoren *
// denne swap-funksjonen kan kalles med følgende kode
// for hhv. verdi-variabler og peker-variabler.

int main(){
    int x = 5;
    int y = 6;
    mySwap(&x, &y);

    int *px = new int(20);
    int *py = new int(10);
    mySwap(px, py);
}
```

**OPPGAVE 2 (40 %): Klasser og operatorer**

I denne oppgaven skal du lage klasser for kort i en kortstokk. Et kort har en farge (spar, hjerter, ruter, kløver) og en verdi (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, knekt, dame, konge, ess). Du skal implementere en supertype kalt **`Card`** og subtypene **`Spade`**, **`Heart`**, **`Diamond`**, **`Club`** (hhv. spar, hjerter, ruter kløver). Lovlige kortverdier skal implementeres ved hjelp av en enumeration (**`enum`**) kalt **`Value`** bestående av verdiene: **`TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN, JACK, QUEEN, KING, ACE`**.

a) Deklarer enum-typen **`Value`** og vis hvordan du kan sørge for at det er samsvar mellom konstanten og heltallsverdien for denne; dvs. TWO med verdien 2, THREE med verdien 3, osv, JACK er 11, QUEEN er 12, KING er 13 og ACE er 14.

```
enum Value {TWO = 2, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE,
TEN, JACK, QUEEN, KING, ACE};
```

b) Implementer supertypen **`Card`** og en eller flere av subtypene **`Spade`**, **`Heart`**, **`Diamond`** og **`Club`**. Alle kortene skal ha en innkapslet medlemsvariabel for verdi (av typen **`Value`**) og kortverdien skal kun settes av konstruktøren. Vis fornuftig bruk av arv og konstruktører og forklar koden din kort.

```
class Card{
   private:
      Value value;
      // viktig å bruke private variabel

   public:
      Card(Value value): value(value){};
      // her kan vi bruke initialiseringsliste og da blir
      // blir konstruktørimpl. Tom.
      Value getValue(){return value;};
};

class Heart: public Card{
public:
   Heart(Value value): Card(value){};
   // her er det viktig å kalle superklassens
   // konstruktør i initialiseringslista
};

class Diamond: public Card{
public:
   Diamond(Value value): Card(value){};
};

osv. for Spade og Club
```

c) Du ønsker å kunne skrive ut en tekstreng bestående av verdi og farge for hvert kort ved hjelp av operatoren **cout** og operatoen **<<**. Kortverdien skal skrives ut som heltall for kortene 2-10 og teksten ”Jack”, ”Queen”, ”King” og ”Ace” for billedkortene og essene. Kortfarge skal skrives ut med teksten ”Spades”, ”Heart”, ”Diamonds” og ”Clubs” (se eksempelet under). Implementer operatoren og nødvendig tillegsfunksjonalitet i klassene slik at du oppnår utskriften som forklart i koden over. NB! Her er vi ute etter både implementasjon av operatoren og bruk av virtuelle metoder. Gi en kort forklaring på koden din (hvorfor du velger å gjøre det slik du gjør).

Oppgaven inneholder to utfordringer.

Det ene er at du må kunne overlagre << operatoren riktig for Card-klassen

Det andre er at du må bruke en virtuell metode for å få skrevet ut klasse-spesifikke delen av tekststrengen. I tillegg må du kunne skrive ut tekstrepresentasjonene av Jack, Queen, King og Ace.

```
class Card{
    private:
        Value value;

    public:
        Card(Value value): value(value){};
        Value getValue(){return value;};
        virtual string getSuiteString() = 0;
};

class Heart: public Card{
public:
    Heart(Value value): Card(value){};
    string getSuiteString(){return "Hearts";};
};

class Diamond: public Card{
public:
    Diamond(Value value): Card(value){};
    string getSuiteString(){return "Diamonds";};
};


// denne operatoren må implementeres som ikke-medlem siden ostream&
// er første parameter. Siden vi kun benytter oss av public funksjoner trenger
// den ikke å deklareres som friend i Card-klassen

ostream& operator<<(ostream& outputStream, Card& card){
    if (card.getValue() < JACK){
        outputStream << card.getValue();
    }else if (card.getValue() == JACK){
        outputStream << "Jack";
    }else if (card.getValue() == QUEEN){
        outputStream << "Queen";
    }else if (card.getValue() == KING){
        outputStream << "King";
    }else if (card.getValue() == ACE){
        outputStream << "Ace";
    }
    outputStream << " of " << card.getSuiteString() << endl;
    return outputStream;
};
```

d) Implementer et eksempel på en av sammenligningsoperatorene slik at du kan teste om et kort er større enn, mindre enn eller er lik et annet kort. Også i denne oppgaven er vi ute etter både operatorimplementasjonen og annen funksjonalitet som er nødvendig for å få dette til å fungere.

Et kort sammenlignes både på grunnlag av verdi og farge. Hvis kortene har ulik verdi er det verdien som teller, hvis de har samme verdi er det fargen som avgjør:
spar **>** hjerter **>** ruter **>** kløver.

```
Club c1(TWO);
Heart c2(JACK);

if (c1 > c2){
    cout << c1 << "er høyere enn" << c2 << endl;
}else{
    cout << c1 << "er ikke høyere enn" << c2 << endl;
}
```

```
enum Suite {CLUB = 1, DIAMOND = 2, HEART = 3, SPADE = 4};

class Card{
    private:
        Value value;

    public:
        Card(Value value): value(value){};
        Value getValue(){return value;};
        virtual string getSuiteString() = 0;
        virtual Suite getSuiteValue() = 0;
        bool operator==(Card& card);
        bool operator<(Card& card);
        bool operator>(Card& card);
        bool operator<=(Card& card);
        bool operator>=(Card& card);
};

class Heart: public Card{
public:
    Heart(Value value): Card(value){};
    string getSuiteString(){return "Hearts";};
    Suite getSuiteValue(){return HEART;};
};

…….

bool Card::operator==(Card& card){
    return (this->getSuiteValue() == card.getSuiteValue()
            && this->value == card.value);
};


bool Card::operator>(Card& card){
    if (this->value > card.value){
        return true;
    }else if (this->value < card.value){
        return false;
    }else{
        return this->getSuiteValue() > card.getSuiteValue();
    }
};

Osv. for evt. Andre operatore
```

## OPPGAVE 3 (30 %): Set av kort

Du skal implementere en klasse kalt **`Cards`** som skal kunne brukes til organisere et sett av kort-objekter (Card-objekter fra forrige oppgave). I denne oppgaven er det viktig å skjønne kravene, og du skal kun implementere noen av egenskapene til klassen.

**`Cards`**-klasse skal ha funksjoner for å legge til kort og ta ut kort fra **`Cards`**. I tillegg skal klassen ha funksjoner for å organisere kortene i tilfeldig rekkefølge (stokke om) eller sortere kortene. **`Cards`**-objekter kan enten inneholde et begrenset antall Card-objekter eller et ubegrenset antall (avhengig av hvilken konstruktør som kalles). Generelle krav til klassen er at rekkefølgen til kortene er signifikant. Når du legger til kort skal du kunne hente dem ut i samme rekkefølge som de ble lagt til, og hvis du stokker kortene får de en ny rekkefølge. Hvis kortene sorteres skal de etterpå være i sortert rekkefølge. Cards-klassen skal også kunne støtte unike sett av kort (alle kort må være forskjellig) eller ikke-unike sett av kort (kortene

kan være like - for eksempel hvis de kommer fra 2 eller flere kortstokker). En parameter i konstruktør-kallet avgjør om instansen sjekker på unikhet eller ikke. Følgende kode er en ufullstendig deklarasjon av Cards-klassen og i deloppgavene under er det spesifisert hva du skal svare på og gjøre i denne oppgaven (kun deler av klassen skal implementeres).

```
class Cards{

public:
   Cards(bool unique, int maxsize);
   Cards(bool unique);
   void add(....) throw (UniqueException, MaxSizeException);
   void sort;
   void shuffle;
};
```

a) I implementasjonen av klassen **`Cards`** har du bruk for noen medlemsvariabler. Forklar hvilke variabler du trenger, hvilke datatyper du bruker og bruk av pekervariabler kontra verdi-variabler. Ta hensyn til kravene som er beskrevet i introduksjonen til denne oppgaven og begrunn dine valg.

> Løsningsforslaget til denne oppgaven er kun et eksempel på hvordan dette kan løses. Vi trenger en bools variabel som kan settes til true/false ihht om det skal være unikt sett eller ikke. I tillegg ville jeg ha benyttet en boolsk variabel for å sette om objektet har maxsize eller ikke (som beskrevet i deloppgave b) samt en int – variabel for maxsize.
>
> Som datastruktur for Card-objektene kan vi velge blant flere alternativer: lenket liste (egendefinert datastruktur) , dynamisk array eller noen av template-klassene i STL (for eksempel vector eller list). Hver av disse har sine fordeler og ulemper. I denne klassen ønsker vi en posisjonsbasert datastruktur hvor vi enkelt kan kontrollere rekkefølgen, legge til på slutten, fjerne osv. Bruk av STL-klassene er den ”enkleste” løsningen siden disse vil kreve minst koding i klassen: vector<*Card> er en grei løsning (selv om det kanskje vil gi deg noen utfordringer mht. fjerning av kort som beskrevet i oppgave c). Felle krav uansett datatype du velger er at du lagrer pekere, at størrelsen kan varieres, samt at du kan lagre duplikater (i tilfellet ikke-unikt).

b) Implementer konstruktørene til klassen. **`unique`**-paramenteren brukes for å spesifisere om **`Cards`**-objektet kan inneholde 2 eller flere like kort (samme farge og verdi) eller om **`Cards`**-klassen kun skal kunne inneholde unike kort (alle kort skal ha forskjellig farge og verdi). Hvis konstruktøren som har en **`maxsize`**-parameter benyttes skal objektet som instansieres kun inneholde opp til et maks antall kort. Hvis konstruktøren uten **`maxsize`** benyttes er det ingen begrensinger på antall kort som kan lagres.

```
class Cards{
    private:
        bool hasMax;
        int maxsize;
        bool unique;
        vector<Card*> cards;
    public:
        Cards(bool unique, int maxsize): hasMax(true), maxsize(maxsize), unique(unique){};
        Cards(bool unique): hasMax(false), maxsize(-1), unique(unique){};
        //Viktig i denne oppgaven å sette verdiene for maxsize, hasMax og unique
        //For sikkerhets skyld er det greit å sette maxsize til et "kjent" tall også
        //for instanser som ikke har maxsize: bruker -1 til dette
};
```

c) Implementer **add**-funksjonen. I denne oppgaven er vi spesielt interessert i unntaksmekanismen. For Cards-objekter med **maxsize** skal unntak av typen **MaxSizeException** kastes hvis du prøver å legge til flere kort en lovlig. For Cards-objekter som er instansierte med **unique** = **true** skal det kastes et unntak av typen **UniqueException** hvis du prøver å legge til et kort som har samme farge/verdi som et annet kort som allerede finnes.

```
class UniqueException{};
class MaxSizeException{};

bool Cards::contains(Card* c){
    // hjelpemetode for å sjekke om et kort finnes fra før i et Cards-objekt
    // kan også bruke STL-algoritme til dette , men her er en enkel variant
    for (unsigned int i = 0; i < cards.size(); i ++){
        if (*c == *(cards[i])){ // bruker sammenligningsoperator fra oppgave 2.
            return true;
        }
    }
    return false;
}

void Cards::add(Card*c) throw (UniqueException, MaxSizeException){
    if (hasMax){
        if (cards.size() >= maxsize){
            throw MaxSizeException();
        }
    }
    if (unique && contains(c)){
        throw UniqueException();
    }
    cards.push_back(c);

}
```

d) Implementer en mekanisme for å hente ut og fjerne kortene fra et **`Cards`**-objekt ett for ett. Mekanismen skal kunne brukes til å gå igjennom alle kortene i den rekkefølgen de til enhver tid er organisert i. Mekanismen trenger ikke ta hensyn til stokking og sortering siden vi overlater til de som bruker klassen å sørge for at de ikke stokker eller sorterer mens de går igjennom sekvensen av kort. Her er vi primært ute etter hvilke(n) funksjon(er) du vil basere deg på deg og det holder at du skriver funksjonsdeklarasjonene og implementer/forklarer basisprinsippet ditt.

Her er vi ute etter en eller flere funksjoner du kan benytte for å hente ut og fjerne kort fra Cards (derfor er det kalt en "mekanisme"). I praksis kan dette være litt vanskelig å implementere og løsningen du velger vil avhenge av hvilken datastruktur du har brukt for card-objektene. For vector<Card*> er det vanskelig å bruke både iterator og []-operatoren hvis du ønsker å hente ut og fjerne i en og samme funksjon fra begynnelsen av; husk at hvis du fjerner en objektreferanse vil indeksene som kommer etter bli endret og størrelsen vil bli endret!

En enkelt workaround ved bruk av vector er å benytte pop_back(), og da henter vi ut ett og ett kort fra enden av vector i stedet. Her bruker vi NULL for å teste på om det er flere elementer igjen.
NB! Oppgaven kan løses på mange måter…….

```
Card* Cards::get(){
   if (cards.empty()){
      return NULL;
   }else{
      Card* c = cards.back();
      cards.pop_back();
      return c;
   }
}

I main:

   Card *c = cards.get();
   while(c != NULL){
      cout << *c;
      c = cards.get();
   }
```

e) Du har bestemt deg for at Cards-klassen er en nyttig type klasse som kan generaliseres og benyttes til mange forskjellige typer objekter og ønsker å lage en template-klasse kalt MyCollectionClass. Vis med kode og forklar hvordan du kan lage en template-klasse som har de samme funksjoner som Cards-klassen. Forklar eventuelle endringer du må gjøre i koden og mulige begrensinger som kan finnes i bruken av denne template-klassen.

Vi må lage oss en template-klasse hvor Card parameteriseres. I tillegg vil det være naturlig å endre noen av metodenavnene til mer generelle navn ;-)

Generelt: vi må deklarere en template og erstatte alle referanser til den variable typen med bokstaven vi har deklarert for variabel type. Hvis vi benytter typespesifikke operatorer (for eksempel == ) er det en forutsetning at klassene eller andre typer vi bruker denne templaten til, har overlagret slike operatorer. Ellers vil det ikke kompilere.

```
template <class T>
class MyCollectionClass {
private:
        bool hasMax;
        unsigned int maxsize;
        bool unique;
        vector<T*> coll;
public:
        MyCollectionClass(bool unique, int maxsize):
                    hasMax(true), maxsize(maxsize), unique(unique){};
        MyCollectionClass(bool unique):
                    hasMax(false), maxsize(-1), unique(unique){};
        void add(T *c) throw (UniqueException, MaxSizeException);
        bool contains(T *c);
        T *get();
}


T *MyCollectionClass::get(){
    if (coll.empty()){
        return NULL;
    }else{
        T *c = coll.back();
        coll.pop_back();
        return c;
    }
}
```