NTNU
Institutt for datateknikk
og informasjonsvitenskap

**Eksamensoppgave i**

# TDT4102 - Prosedyre- og objektorientert programmering

## Tirsdag 25. mai 2010

## Kontaktperson under eksamen: Trond Aalberg (97631088)

***Eksamensoppgaven er utarbeidet av Trond Aalberg***
***og kvalitetssikret av Hallvard Trætteberg***

**Språkform:** Bokmål

**Tillatte hjelpmidler:** Walter Savitch, Absolute C++ eller Lyle Loudon, C++ Pocket Reference

**Sensurfrist:** Tirsdag 15 juni.

## Generell introduksjon

Les gjennom oppgavetekstene nøye og finn ut hva det spørres om. Noen av oppgavene har lengre forklarende tekst, men dette er for å gi mest mulig presis beskrivelse av hva du skal gjøre.

All kode skal være C++.

Dersom du mener at opplysninger mangler i en oppgaveformulering, gjør kort rede for de antagelser og forutsetninger som du finner det nødvendig å gjøre. Hver enkelt oppgave er ikke ment å være mer krevende enn det som er beskrevet.

Oppgavesettet er arbeidskrevende og det er ikke foreventet at alle skal klare alle oppgaver innen tidsfristen. Disponer tiden fornuftig!

Oppgavene teller med den andelen som er angitt i prosent. Den prosentvise uttellingen for hver oppgave kan likevel bli justert ved sensur. De enkelte deloppgaver kan også bli tillagt forskjellig vekt.

## Oppgave 1: Grunnleggende programmering (15%)

a) Hva skrives ut av følgende kode:

```
int a = 4;
int b = 5;
cout << "1) = " << a / b << endl;

int c = 4;
int d = 5;
cout << "2) = " << c % d << endl;

bool e = true;
bool f = false;
cout << "3) = " << (!e && f) << endl;

int g = 8;
int h = g++;
cout << "4) = " << g << ", " << h << endl;

int* i = new int(5);
int* j = new int(10);
swap(i, j);
cout << "5) = " << *j << endl;
```

b) Implementer en funksjon **`bool match(string word, string pattern)`** som kan brukes for å teste om et ord matcher et søkeuttrykk. Søkeuttrykket (pattern-argumentet) kan bestå av bokstavene a-å/A-Å eller tegnet '.' som representerer en vilkårlig bokstav.
**`match("testing", "t...i")`** skal gi **`true`** fordi ordet har 't' som første bokstav etterfulgt av tre vilkårlige bokstaver før 'i'.
Tips: husk at det er mulig å bruke indeks-operatoren på string-variabler, og du kan anta at et ord bare består av bokstaver.

c) Den rekursive funksjonen under er ment å skulle skrive ut tegnene i en C-streng baklengs. Hvis du kaller funksjonen med strengen"*Dette er en test*" er det meningen at utskriften skal bli "*tset ne re etteD*". Finn feil i koden og skriv riktig kode i svaret ditt.

```
void recursive(char* t){
  if (*t != '\0'){
    recursive(t++);
  }
  cout << *t;
}
```

## Oppgave 2: Programmering med funksjoner (35%)

I denne oppgaven skal du lage et program som leser innholdet i et tekstdokument (en tekstfil) og skriver ut en rapport (til en annen tekstfil) med informasjon om antall linjer i tekstdokumentet, samt hvilke bokstaver og ord som er brukt. Du kan anta at tekstdokumentet inneholder vanlige setninger hvor det er brukt skilletegn som mellomrom, komma og punktum. Du trenger ikke ta hensyn til andre tegn.

Programmet skal fungere på følgende måte:
Ved oppstart skal programmet spørre etter navnet på tekstfilen som skal leses. Hvis det ikke finnes en fil med dette navnet skal programmet skrive ut en feilmelding og avslutte. Hvis filen finnes skal programmet spørre brukeren etter navnet på ei fil som rapporten skal skrives til.

Innholdet i rapporten skal være:
1) Antall linjer i tekstdokumentet som er lest.
2) Ei liste over bokstavene a-z og hvor mange ganger hver bokstav forekommer i fila som er undersøkt (f.eks. a:10 b:12 c:0 d:2 osv). Store og små bokstaver håndteres som samme bokstav.
3) En sortert liste over alle ordene som er brukt og antall ganger hvert ord er brukt.

VIKTIG: Dette er en åpen oppgave hvor du først og fremst skal vise bruk av funksjoner og variabler samt kjennskap til nyttige typer og funksjoner i C++ biblioteket. Du trenger IKKE lage en fullstendig implementasjon av alle egendefinerte funksjonene, men skal vise nok til at den som leser koden din kan forstå hvordan du har planlagt at programmet skal fungerer. Utover beskrivelsen over står du fritt i hvordan programmet oppfører seg og hvordan rapporten skal formatteres.

Du skal i denne oppgaven svare som en samlet implementasjon. Deloppgavene lister opp noen av de spesifikke tingene vi kommer til å se etter i implementasjonen din.

a) Hvilke C++ biblioteker du benytter (vis ved hjelp av include-statements).

b) Hvilke datatyper du bruker for å lagre informasjon internt i programmet og hvordan du benytter disse. Du bør velge typer som gjør det enklest mulig å programmere løsningen).

c) Innhenting av input fra bruker (filnavnene som bruker skal skrive inn).

d) Hvilke egendefinerte funksjoner du benytter for å gjøre programmet enkelt å jobbe med (koding), enkelt å vedlikeholde (rette feil), samt enkelt å lese/forstå for andre.

e) Lesing av fil samt skriving til fil (ved hjelp av funksjoner eller ved å overlagre operatorer).

f) Riktig bruk av løkker, forgreininger og betingelser for å kontrollere programflyten.

## Oppgave 3: Klasser (25%)

I denne oppgaven skal du lage en klasse kalt **`Timer`**. Denne klassen skal kunne brukes for å måle tidsbruk omtrent på samme måte som du vil bruke en Stoppeklokke. Du bør lese igjennom alle deloppgavene for å få en fullstendig oversikt over hvordan klassen skal virke.

Timer-klassen skal benytte funksjonen **`int clock()`** fra **`ctime`**-biblioteket for å beregne tidsbruk[1]. Denne funksjonen returnerer antallet taktslag som har gått siden et program startet (hvor mange ganger prosessoren har "tikket" siden oppstart av programmet). Selve tidtakingen styres med eksplisitte kall til Timer-klassens medlemsfunksjoner **`void start()`** og **`void stop()`**. Disse skal implementeres slik at man kan starte og stoppe tidtakingen flere ganger etter hverandre for å akkumulere tidsbruk. Klassen skal også ha medlemsfunksjoner som returnerer tidsbruken som henholdsvis *minutter*, *sekunder* eller *millisekunder* (tusendels sekunder). Vi kan bruke konstanten **`CLOCKS_PER_SEC`** fra **`ctime`**-bibliotek for å regne om fra taktslag til sekunder.

Klassen skal implementeres som beskrevet i deloppgavene, men du velger selv om du vil presentere kode under hver deloppgave eller som en helhetlig og samlet implementasjon. Velger du en samlet implementasjon bør du kommentere koden slik at det går tydelig fram hvilke deler du har svart på.

a) Hvilke medlemsvariabler (en eller flere) må klassen ha for å kunne ta vare på tidsbruk og annen informasjon som er nødvendig.

b) Hva er innkapsling (generelt) og hvordan har du brukt innkapsling i Timer-klassen?

c) Implementer en av følgende medlemsfunksjoner: En funksjon som returnerer tidsbruken som hele minutter, en funksjon som returnerer tidsbruken som hele sekunder eller en funksjon som returnerer tidsbruken som millisekunder (tusendels sekunder) .

d) Hva er formålet med en klasses konstruktør(er)? Implementer en egnet konstruktør for Timer-klassen.

e) Implementer medlemsfunksjonene **`start`** og **`stop`**. Disse skal implementeres slik at man kan starte og stoppe tidtakingen flere ganger for å akkumulere tidsbruk. Det skal derfor være mulig å gjøre funksjonskall-sekvensen **`t.start(); t.stop(); t.start(); t.stop();`**
Dette skal gjøre at **`t`** lagrer tiden mellom første **`start`** og **`stop`** og etterpå legger til tiden mellom andre **`start`** og **`stop`**.

f) I praksis er kall til **`stop`** kun meningsfullt hvis tidtakingen allerede er startet (det har vært utført et kall til start). Kall til **`start`** er kun meningsfullt hvis tidtakingen ikke allerede er startet. Dette kan gi potensielt gi problem f.eks. hvis man gjør kallsekvensen **`t.start(); t.start(); t.stop(); t.stop();`**. Forklar hvordan du velger å håndtere dette (dette er en åpen oppgave og vi legger primært vekt på gode argumenter for ditt valg).

g) Overlagre en (fritt valgt) sammenligningsoperator som medlem av klassen. Operatoren skal sammenligne tidsbruken som 2 Timer-objektene representerer. Du kan anta at operatoren bare benyttes når tidtakingen i begge objekter er "stoppet" opp. Objektene skal sammenlignes mht. antall taktslag.

---

1. De som har vært borte i denne funksjonen tidligere vil kanskje legge merke til at vi har forenklet litt.

## Oppgave 4: En dynamisk todimensjonal tabell (25%)

I denne oppgaven skal du lage en objektorientert datastruktur som ligner litt på en todimensjonal array, men hvor du dynamisk kan legge til nye rader av forskjellig lengde (mens programmet kjører). Datastrukturen skal være basert på en lenket liste hvor radene er noder med hver sin peker til et dynamisk allokerte array. Det er med andre ord ikke lov å bruke vector eller andre klasser fra Standard Template Library. Se vedlegget for en illustrasjon.

Deklarasjonen av klassene **`DynamicMultiArray`** og **`RowNode`** er vist under (men du må selv bestemme om det er behov for andre variabler eller funksjoner). Funksjonen **`addRow`** brukes for å legge til en og en rad. Medlemsvariablene **`first`** og **`last`** peker hhv. til første og siste element i den lenkede listen av rader. Enkelt-verdier kan leses med funksjonen **`at`** hvor første argument (**`row`**) er raden du skal lese fra (nummerert fra 0 og oppover), og andre argument (**`column`**) er indeksen til elementet du skal lese i denne raden.

```
class DynamicMultiArray{

private:
   RowNode *first;
   RowNode *last;
public:
   DynamicMultiArray();
   ~DynamicMultiArray();
   void addRow(int row[], int size);
   int at(int row, int column);
};
```

```
class RowNode{

private:
   int *row;
   RowNode* next;
   int rowsize;
public:
   RowNode(int row[], int size);
   ~RowNode();
   friend class DynamicMultiArray;
};
```

a) Implementer konstruktøren til **`RowNode`**. Konstruktøren skal initialisere objektene riktig og kopiere innholdet i row-argumentet over i en dynamisk allokert array som medlemsvariabelen row peker til.

b) Implementer konstruktøren til **`DynamicMultiArray`**.

c) Implementer medlemsfunksjonen **`addRow`**. Nye rader skal legges til på slutten av lista.

d) Implementer medlemsfunksjonen **`at`**. Parametrene **`row`** og **`column`** er her indekser tilsvarende syntaksen [row][column] for todimensjonale tabeller. Funksjonen skal returnere verdien som finnes på denne posisjonen. Hvis denne posisjonen ikke finnes skal funksjonen kaste et unntak av typen **`InvalidIndex`** (du må også deklarere denne typen).

e) Hvordan må funksjonen **`at`** deklareres hvis du skal kunne bruke denne til å endre på enkeltelementer i tabellen slik som vist i denne setningen **`d.at(3,3) = 5;`**
(hvor **`d`** er en variabel av typen **`DynamicMultiArray`**)

f) Hva betyr følgende linje i RowNode: **`friend class DynamicMultiArray`**?

g) Hvordan kan du deklarere at **`addRow`** ikke endrer på verdiene til argumentet **`int row[]`**?

h) Implementer destruktoren i begge klasser slik at alt minne som er allokert blir frigjort når et **`DynamicMultiArray`**-objekt slettes (enten automatiske variabel som går ut av scope eller dynamisk allokerte variabel som eksplisitt slettes med delete).

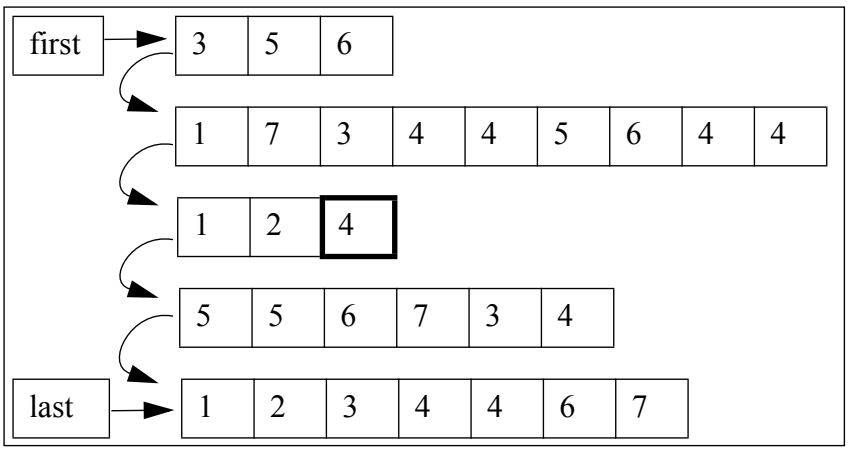
i) Hvilken mekanisme kan du bruke for å lage en generisk versjon av **`DynamicMultiArray`** (slik at du kan ha **`DynamicMultiArray`** variabler for å lagre andre typer enn int);

# VEDLEGG

Eksempel som illustrerer bruken av DynamicMultiArray.

```
DynamicMultiArray *d = new DynamicMultiArray();

int a[3] = {3, 5, 6};
d->addRow(a, 3);

int b[9] = {1, 7, 3, 4, 4, 5, 6, 4, 4};
d->addRow(b, 9);

int c[3] = {1, 2, 4};
d->addRow(c, 3);

int d[6] = {5, 5, 6, 7, 3, 4};
d->addRow(d, 6);

int e[7] = {1, 2, 3, 4, 4, 6, 7};
d->addRow(e, 7);
```

//datastrukturen etter at radene er lagt til med addRow:




```
cout << d->at(2,2) << endl;
//skriver ut tallet i ruten med fet linjetype (oppgave 4d)

d->at(2,2) = 10;
//endrer innholdet i elementet som er merket med fet linjetype (oppgave 4e)

delete d;
//sletter d-variabelen og alt minne som brukes
```