NTNU
Institutt for datateknikk
og informasjonsvitenskap

Kontinuasjonseksamen i
TDT4102 - Prosedyre- og objektorientert programmering

Faglig kontakt under eksamen: Trond Aalberg

Tlf: 97631088

Eksamendato:	**12. august 2015**
Eksamenstid:	**15-19**
Hjelpemiddelkode:	**C: Spesifiserte trykte og håndskrevne hjelpemidler tillatt.**
	Walter Savitch, Absolute C++ eller
	Kyle Loudon, C++ Pocket Reference.
Målform/språk:	**Bokmål / Nynorsk**
Antall sider:	**6 sider inklusive forside**

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 *"vis"* eller "*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 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 bruke navnerom, 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. Det er ikke noen systematisk sammenheng mellom vanskelighetsgrad og nummerering av deloppgavene.

Hoveddelene av eksamensoppgaven teller med den andelen som er angitt i prosent. De enkelte deloppgaver vil bli tillagt forskjellig vekt basert på vanskelighetsgrad og arbeidsmengde.

Oppgave 1: Kodeforståelse og funksjoner (40%)

a) Hva skrives ut av følgende kode?

```
int a = 4;
int b = 9;
cout << "1) = " << a / b << endl;

int c = 6;
int d = 4;
cout << "2) = " << c % d << endl;

cout << "3) = " << static_cast<int>(11.9/4) << endl;

bool e = true;
bool f = false;
cout << "4) = " << boolalpha << !(e && f) << endl;
//boolalph is a modifier that causes a boolen value to be
//printed as the string "true" or "false"

int g = 8;
int h = ++g;
cout << "5) = " << g++ << ", " << h << endl;

int* i = new int(5);
int* j = new int(10);
swap(i, j);
cout << "6) = " << *j << endl;

int* k = new int(5);
int* l = new int(10);
swap(*k, *l);
cout << "7) = " << *l << endl;
```

```
1) = 0
2) = 2
3) = 2
4) = true
5) = 9, 9
6) = 5
7) = 5
```

b) Hva er feil med følgende funksjon og hvordan kan den implementeres (gitt at header skal beholdes som den er)?

```
char* createAlphabet(char start, int length){
    char alphabet[length]; //bør ha char* alphabet = new char[length]
    for (int i = 0; i < length; i++){
        alphabet[i] = start++;
    }
    return alphabet;
}
Vi prøver å lage en tabell hvor størrelsen skal bestemmes runtime.
Dette er lov på enkelte kompilatorer og dermed ikke den verste feilen.
Det egentlige problemet er at vi returnerer en peker til denne automatisk
allokerte tabellen, men denne frigis når den går ut av scope.
```

c) Implementer funksjonen **`bool isAnagram(const string& a, const string& b)`** som sjekker om to tekststrenger er anagram av hverandre. Anagram er et ord, navn eller uttrykk som er blitt satt sammen ved å stokke rundt på bokstavene i et annet ord eller uttrykk (man ser vanligvis bort fra mellomrom og tegnsetting mellom bokstavene). Eksempler: "Kama Sutra" = "Austmarka", "Rivaliserende" = "Avleirende ris", "William Shakespeare" = "I am a weakish speller".
*Dette kan gjøres ved å lage seg kopier av strengene hvor du har fjernet tegn som ikke er bokstaver f.eks. med hjelp av biblioteksfunksjonen **isalpha** og har konvertert til små bokstaver med **tolower**. Et string-objekt er egentlig bare en sekvens av bokstaver og bokstavene kan sorteres f.eks. ved funksjonskallet **sort(str.begin(), str.end())**. Deretter kan du sammenligne for å sjekke om string-argumentene a og b er anagrammer av hverandre.*

```
bool isAnagram(const string& s1, const string& s2){
//implementert omtrent som i oppgavebeskrivelsen, kan også bruke vector<char>.
//Annen løsning er map<char, int> og da trenger vi ikke eksplistt sortering
    string temp1;
    string temp2;
    for (char c: s1){
        if (isalpha(c)){
            temp1 += tolower(c);
        }
    }
    for (char c: s2){
        if (isalpha(c)){
            temp2 += tolower(c);
        }
    }
    sort(temp1.begin(), temp1.end());
    sort(temp2.begin(), temp2.end());
    return temp1 == temp2;
}
```

d) Implementer funksjonen **`bool isPalindrome(const string& str)`** som sjekker om en tekststreng er et palindrom. Et palindrom er et ord, uttrykk eller tall som gir samme resultat enten det leses fra høyre eller venstre (man ser vanligvis bort fra mellomrom og tegnsetting mellom bokstavene). Eksempler på palindrom er: "dvd", "rør", "redder", "trenert", "Agnes i senga.", "Skal varg ete gravlaks?".

```
bool isPalindrom(const string& s){
    if (s.size() == 0)
        return false;
    int first = 0;
    int last = s.size() - 1;
    while (first <= last){
        while (!isalpha(s[last]) && first <= last)
            last--;
        while (!isalpha(s[first]) && first <= last)
            first++;
        if (tolower(s[first]) != tolower(s[last]))
            return false;
        first++;
        last--;
    }
    return true;
}
//Det står også en løsning i boka som er basert på bruk av diverse delfunksjoner.
```

De som kun skrev av boka kastet bort mye tid og fikk mindre score enn de som laget en egen og mer kompakt (les mindre tidkrevende) løsning.

e) Implementer en funksjon **`void printWordStatistics(const string& filename)`** som leser ei tekstfil og skriver ut (til **`cout`**) ei sortert liste av alle unike ord sammen med antallet forekomster av ordet i fila. Eksempel:

```
alt: 4
bra: 2
det: 3
er:  6
```

```
void printWordStatistics(const string& filename){
    ifstream input(filename);
    string word;
    map<string, int> wordmap;
    while (input >> word){
        wordmap[word]++;
    }
    for (auto it: wordmap){
        cout << it.first << ": " << it.second << endl;
    }
     //leseløkka sjekker implisitt om fila er åpnet, hvis ikke vil
     //map'en være tom og ingenting skrives ut
}
```

f) Implementer funksjonen **`double mean(double arr[], int size)`** som regner ut det aritmetiske gjennomsnittet av verdiene i tabell arr. Det aritmetiske gjennomsnittet finner du ved å dele summen av alle verdiene med antallet verdier. Parameteren **`size`** er størrelsen på tabellen (antallet verdier i tabellen).

```
double mean(int arr[], int size){
    int sum = 0;
    for (int i = 0; i < size; i++){
        sum += arr[i];
    }
    return sum / static_cast<double>(size);
} //eller deklarer sum som double og utelat casting;
```

g) Implementer 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). 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.

```
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;
    }
}
```

h) Implementer funksjonen **`double round(double num, int precision)`** som skal returnere verdien **`num`** avrundet til antallet desimaler angitt med **`precision`**. Eksempelvis skal round(0.886, 2) gi 0.89 og round(-0.788, 1) skal gi -0.8. Funksjonen under viser hvordan et flyttall kan avrundes til et heltall (med riktig avrunding) og denne kan du basere deg på i løsningen din (denne er det basert på implisitt kasting pga. returtypen).

```
int round(double num){
    if (num >= 0.0){
        return num + 0.5;
    }else{
        return num - 0.5;
    }
}
```

```
//Her trikser vi og gjør om num til heltall ved å opphøye til 10**precision
//Da sitter vi igjen med et heltall som kan avrundes med int-versjonen av round
//For å gjøre om til flyttal deler vi med 10**precision (10 opphøyd i prec.).
//Vi kan bruke biblioteksfunksjonen pow eller anta vår egen funksjon
double round(double num, unsigned int precision){
        int x = power(10, precision);
        return (round(num * x)) / (double) x;
}
```

```
NB! Hvis du skal prøve ut denne bør du unngå bruk av "using namespace std" ellers
vil du få problemer med overlagring av round(double) siden den finnes fra før i
biblioteket.
```

i) I medlemsfunksjonen **`func`** i klassen **`Foo`** under er nøkkelordet **`const`** brukt på tre plasser. Forklar hva effekten og hensikten er for hver enkelt av disse.

```
class Foo{
  private:
    set<string> dummyvar;
  public:
    const (1) string& (4) func (const (2) string& (5) p) const (3);
};
//const (1) string& (4) func (const (2) string& (5) p) const (3);
1: returverdien er const og kan ikke endres (via returverdien)
2: parameter er const og kan ikke endres inne i funksjonen
3: spesifiserer at funksjonen ikke endrer noen medlemsvariabler
```

j) I samme funksjon (medlemsfunksjonen **`func`** i oppgaven over) er tegnet **`&`** brukt to plasser. Hva spesifiserer dette tegnet og forklar effekt og hensikt.

```
4: return-by-reference
5: call-by-reference
I begge tilfeller sendes referanse, det gjøres ingen kopiering, vi ønsker å kunne
endre variabler innenfra, eller ønsker å kunne oppdatere via retur-variabelen,
eller ønsker å unngå kopiering.
```

Oppgave 2: Lenket liste av dynamisk allokerte tabeller (40%)

I denne oppgaven skal du lage en objektorientert datastruktur som ligner litt på en todimensjonal array av **int**, men hvor du dynamisk kan legge til nye rader av forskjellig lengde (mens programmet kjører). Datastrukturen skal være basert på en dobbelt-lenket liste hvor nodene er objekter med hver sin dynamisk allokerte tabell. Det er ikke lov å bruke vector eller andre klasser fra biblioteket i denne oppgaven.

En foreløpig deklarasjon av klassene **DynamicMultiArray** og **RowNode** er vist under, men du må selv bestemme om det er behov for andre variabler eller funksjoner samt legge til funksjonene vi ber om i deloppgavene).

```
class DynamicMultiArray{
private:
   RowNode *first;
   RowNode *last;
public:
   DynamicMultiArray();
   ~DynamicMultiArray();
   void add(int size);
   void remove(int row);
   int get(int row, int column);
   void set(int row, int column, int val);
};
```

```
class RowNode{
private:
   int *arr;
   RowNode* next;
   RowNode* prev;
   int size;
public:
   RowNode(int size);
   ~RowNode();
   friend class DynamicMultiArray;
};
```

Hver rad i tabellen er en instans av klassen **RowNode**. Denne klassen har de klassiske **next**- og **prev(ious)**-peker-medlemsvariablene som brukes for å implementere en dobbelt-lenket liste. next skal peke til rad-noden som kommer etter og prev skal peke til rad-noden som kommer før. Siden radene skal være dynamisk allokerte tabeller har vi også en peker **arr** som skal peke til den dynamisk allokerte tabellen hvor dataene i raden skal lagres. I variabelen size lagrer vi størrelsen på raden.

a) Implementer konstruktøren til **RowNode**. Konstruktøren skal sørge for at det allokeres minne til en int-array av størrelsen **size**. Vis bruk av initialiseringsliste.

```
RowNode::RowNode(unsigned int size):
          size(size), arr(new int[size]{}),
          next(nullptr), prev(nullptr){
     //her er alt gjort i initialiseringslista til konst.
     //I oppgaven er det brukt int som parametertype for size
     //for å gjøre signarturene raskere å skrive på eksamen, men ellers er det
     //god skikk å bruke unsigned int for størrelse på array
}
```

b) Implementer destruktøren til **RowNode**.

```
RowNode::~RowNode(){
    delete [] arr;
}
```

c) Hva betyr følgende linje i RowNode: **friend class DynamicMultiArray** og hvilken nytteverdi har dette i implementasjonen av denne datastrukturen?

Det betyr at DynamicMultiArray får tilgang til private deler av RowNode. friend er en mekanisme som brukes for å overstyre private/public-mekanismen. Siden arr er en privat variabel i RowNode kan ikke denne endres/skrives til fra kode utenfor klassen, men siden DynamicMultiArray er "venn" får denne lov til å skrive/lese arr. Dermed kan vi lage en løsning hvor get og set-funksjonalitet bare gjøres tilgjengelig via DynamicMultiArrray og vi oppnår en litt intrikat form for innkapsling.

Klassen **`DynamicMultiArray`** er den overordnede klassen for den lenkede listen og den eneste klassen brukere (den som benytter DynamicMultiArray i et program) trenger å kjenne til. Medlemsvariablene **`first`** og **`last`** peker hhv. til første og siste node (rad). Enkelt-verdier i radene leses og skrives med get- og set-funksjoner hvor første argument (row) er raden du skal lese fra, og andre argument (column) er indeksen til elementet du skal lese i denne raden.

d) Implementer konstruktøren til **`DynamicMultiArray`**. Denne skal opprette en instans uten rader (en tom tabell).

```
DynamicMultiArray::DynamicMultiArray(): first(nullptr), last(nullptr){}
```

e) Implementer medlemsfunksjonen **`void add(int size)`**. Nye rader skal legges til på slutten og size angir størrelsen på raden.

```
void DynamicMultiArray::add(unsigned int size){
    RowNode* rownode = new RowNode(size); //oppretter ny rad
    rownode->prev = last;
    last = rownode;
    if (first == nullptr){
        first = rownode;
    }else{
        last->prev->next = last;
    }
}
```

f) Implementer medlemsfunksjonen **`void remove(int row)`**. Denne brukes til å slette rader fra tabellen hvor parameter **`row`** angir indeksen til raden som skal slettes. Hvis rad med denne indeksen ikke finnes skal funksjonen kaste et unntak av typen **`std::invalid_argument.`**

```
//Baserer oss på en egen funksjon som leter frem raden med en gitt indeks:
RowNode* DynamicMultiArray::find(unsigned int index){
    if (first == nullptr){
        throw std::invalid_argument("Illegal argument, empty list, first ==
nullptr");
    }
    RowNode* current = first;
    for (int i = 0; i < index; i++){
        current = current->next;
        if(current == nullptr){
            throw std::invalid_argument("Illegal argument, index above");
        }
    }
    return current;
}
```

```
void DynamicMultiArray::remove(unsigned int row){
    RowNode* found = find(row); //exception if not found or list is empty
    RowNode* discard = found;

    if (first == nullptr){
        return; //list is empty, actually not needed because of exception
    }
    if (first == found){
        first = found->next;
    }
    if (last == found){
        last = last->prev;
    }
    if (found->next != nullptr){
        found->next->prev = found->prev;
    }
    if (found->prev != nullptr){
        found->prev->next = found->next;
    }
    delete discard;
}
```

g) Implementer get- og set-funksjonene i **`DynamicMultiArray`** for å lese og endre verdier i tabellen. Parametrene **`row`** og **`column`** er her indekser tilsvarende syntaksen [row][column] for todimensjonale tabeller. Hvis denne posisjonen ikke finnes skal funksjonen kaste unntak av typen **`std::invalid_argument`**.

```
int DynamicMultiArray::get(unsigned int r, unsigned int c){
    RowNode* row = find(r);
    if (c < row->size){
        return row->arr[c];
    }else{
        throw std::invalid_argument("Illegal argument, rowindex above");
    }
}

void DynamicMultiArray::set(unsigned int r, unsigned int c, int val){
    RowNode* row = find(r);
    if (c < row->size){
        row->arr[c] = val;
    }else{s
        throw std::invalid_argument("Illegal argument, rowindex above");
    }
}
```

h) Implementer destruktøren i **`DynamicMultiArray`**.

```
DynamicMultiArray::~DynamicMultiArray(){
    RowNode* curr = first;
    while (curr != nullptr){
        RowNode* next = curr->next;
        delete curr;
        curr = next;
    }
}
//Også mulig å tenke gjenbruk av remove her
```

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).

`Template (og bruk helst en setning som viser at du vet hva Template er)`

Oppgave 3: Minnehåndtering (20%)

Her fortsetter vi med klassene **`DynamicMultiArray`** og **`RowNode`** fra oppgave 2.

a) Implementer en medlemsfunksjon **`void RowNode::resize(int increase)`** for å øke størrelsen på en rad med **`increase`** antall elementer.

`Dette er klassisk kode som dere finner eksempler på i øvingene: lag et nytt array som er større, kopier over fra orginal array til nytt, frigi orginalen og setter peker til å peke på nytt array`

b) Implementer det som er nødvendig for klassene **`DynamicMultiArray`** og **`RowNode`** for å støtte "deep copy". Dvs. at det ved tilordning (eller kall til kopikonstuktør) for objekter av typen **`DynamicMultiArray`** skal lages en fullstendige kopi av tabellen. Du trenger kun implementere/vise funksjoner/operatorer som må legges til eller vesentlig endres.

```
Skisse til løsning: legg til kopikonstruktør i RowNode, men sett denne som private. Vi ønsker at
det bare er DynamicMultiArray som skal kunne bruke denne (noe som fungerer fordi den er venn).

RowNode::RowNode(const RowNode& oth): RowNode(oth.size){
//her kaller vi konstruktøren fra deloppgave 2a)
//merk at vi får egentlig får en delvis kopi (lausunge) siden prev og
//next ikke kopieres. Vi ønsker jo ikke å peke til naboer i orginalen.
    for (int i = 0; i < size; i++){
        arr[i] = oth.arr[i];
    }
}

//Da kan vi implemtere kopikonstruktør og tilordningsoperator for DMA
//og med fordel bruke copy-swap teknikken. Destruktør har vi allerede impelementert

DynamicMultiArray::DynamicMultiArray(const DynamicMultiArray& oth): DynamicMultiArray(){
    RowNode* cur = oth.first;
    while(cur != nullptr){
        //Samme innsettingslogikk som i add
        RowNode* rownode = new RowNode(*cur); //oppretter ny rad med kopikonstruktøren
        rownode->prev = last;
        last = rownode;
        if (first == nullptr){
            first = rownode;
        }else{
            last->prev->next = last;
        }
        cur = cur->next;
    }
}

DynamicMultiArray& DynamicMultiArray::operator=(DynamicMultiArray lhs){
    swap(first, lhs.first);
    swap(last, lhs.last);
    return *this;
}
```

c) Lag alternative versjoner av klassene hvor du implementerer "shallow copy" og "referanse-lignende" oppførsel. Dette betyr at objekter av typen **`DynamicMultiArray`**, som er kopier av hverandre, deler minne/data. Ved tilordning eller kall til kopikonstruktør skal det ikke allokeres nytt minne, men kopien skal peke til samme minne/data som objektet det er kopi av. Dette kalles "referanse-lignende" oppførsel fordi objektene da vil oppføre seg som referanser/pekere. Endringer som gjøres på kopien vil også gjelde for originalen og vice versa. Dette kan implementeres med en delt medlemsvariabel som teller hvor mange objekter som deler minne. Når det opprettes en ny kopi skal variabelen inkrementeres, når en kopi slettes skal variabelen dekrementeres. Hvis variabelen blir 0 betyr det at det ikke er noen kopier igjen og minnet som er i bruk kan frigis.

```
//Her bruker vi en use-teller for å holde styr på hvor mange objekter som peker
til samme. RowNode kan vi bruke som den er, men må implementere DMA litt anner-
ledes. For å gjøre det mulig at samme DMA peker til en liste hvor også første
node pekeren peker til kan endres, må vi bruke peker-til-peker ** for first og
last. Ellers brukes samme teknikk som var eksamensoppgave V2014, se LF til den
oppgaven for en enklere intro til hvordan å implementere shallow copy med refe-
ransetelling.

class DynamicMultiArrayShallowCopy{
private:
    //Må impl som ** fordi både first og last kan endres av medlemsfunksjoner
    RowNode **first;
    RowNode **last;
    //trenger en teller for å telle kopier
    int* use;
public:
    //Konstruktør
    DynamicMultiArrayShallowCopy(){
        use = new int(1);
        first = new RowNode*;
        last = new RowNode*;
        *first = nullptr;
        *last = nullptr;
    }
    //Kopikonstruktør
    DynamicMultiArrayShallowCopy(const DynamicMultiArrayShallowCopy& oth){
        this->use = oth.use;
        ++(*use);
        first = oth.first;
        last = oth.last;
    }
    //Tilordningsoperator iml med copy-swap idiomet
    DynamicMultiArrayShallowCopy& operator=(DynamicMultiArrayShallowCopy lhs){
        swap(use, lhs.use);
        swap(first, lhs.first);
        swap(last, lhs.last);
        return *this;
    }
```

```
    //Destruktør
    ~DynamicMultiArrayShallowCopy(){
        if (--(*use) == 0){
            RowNode* curr = *first;
            while (curr != nullptr){
                RowNode* next = curr->next;
                delete curr;
                curr = next;
            }
            delete use;
            delete first;
            delete last;
        }
    }
    //Ingen vesentlig endringer i andre funksjoner,
    //men må implementere med first og last som **
};
```