NTNU
Kunnskap for en bedre verden

Institutt for datateknikk og informasjonsvitenskap

Eksamensoppgave i TDT4102 – Prosedyre- og objektorientert programmering

Faglig kontakt under eksamen: Lasse Natvig

Tlf.: 906 44 580

Eksamensdato: 10 Juni 2016

Eksamenstid (fra-til): kl. 0900 - 1300

Hjelpemiddelkode/Tillatte hjelpemidler: C: Spesifiserte trykte og håndskrevne hjelpemidler tillatt. Bestemt, enkel kalkulator tillatt. Walter Savitch, Absolute C++

Målform/språk: Bokmål

Antall sider : 9

Ingen vedlegg

Informasjon om trykking av eksamensoppgave
Originalen er:

1-sidig X **2-sidig** ☐

sort/hvit ☐ **farger X**

Kontrollert av:
Berit Hellan

Dato Sign

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!

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 i utgangspunktet med den andelen som er angitt i prosent. Den prosentvise uttellingen for hver oppgave kan likevel bli justert ved sensur basert på hvordan oppgavene har fungert. De enkelte deloppgaver kan også bli tillagt forskjellig vekt.

Oppgave 1 – Kodeforståelse (25 %)

Svar på denne måten på et vanlig svarark:

1a) ... ditt svar her

1b) … ditt svar her … osv.

1a) Hva skrives ut ?

```
int a = 5;
int b = 10 / a--;
int c = 10 % (++b);
cout << (a - 2) << endl;
cout << b << endl;
cout << (c += 3) << endl;
```

1b) Hva skrives ut ?

```
int i = 1, j = 3;
while ((j--) > 0) {
        i *= 2;
}
cout << "First i = " << i << endl;

i = 32; j = 3;
do {
        i /= 2;
} while (j-- > 0);
cout << "Second i = " << i << endl;
```

1c) Func2 er definert i del (i), hva skrives ut av koden i del (ii)?

(i)

```
int Func2(int a, int& b, int* c)
{
        a = 3;
        b = 5;
        *c = 20;
        return (a + b + *c);
}
```

(ii)

```
int i = 0, j = 1;
int *k = new int;
*k = 10;
cout << Func2(i, j, k) << ", ";
cout << i << ", " << j;
cout << ", " << *k;
```

1d) Hva skrives ut ?

```
int a = 2;
switch (a) {
        case 1: cout << "10 ";
                break;
        case 2: cout << "20 ";
        case 3: cout << "30 ";
                break;
        default: cout << "END";
                break;
}
```

1e) Anta at eksakt ett kall på funksjonen **`func1`** utføres. Hvor mange objekter av typen **`Box`** instansieres og hvor mange ganger kalles destruktøren til **`Box`**? Svar med de to tallene.

```
void func1() {
        Box a;
        for (int i = 0; i < 7; i++) {
                Box *c = new Box();
                a = *c;
        }
}
```

1f) (i) Hva skrives ut av koden i Del-2 når klassene er definert som i Del-1 ?

(ii) Hva blir utskriften hvis Animal::toStr() blir deklarert som virtual?"

Del-1

```
class Animal {
protected:
    string name;
public:
    Animal(string name) : name(name) {}
    string toStr() {
         return "Animal: " + name;
    }
};

class Dog : public Animal {
public:
    Dog(string name) : Animal(name) {}
    string toStr() {
        return "Dog: " + name;
    }
};
```

Del-2

```
Animal cat("Garfield");
Dog dog("Lassie");
cout << cat.toStr();
cout << endl;
cout << dog.toStr();
cout << endl;

Animal *aptr = &dog;
Dog *dptr = &dog;
cout << aptr->toStr();
cout << endl;
cout << dptr->toStr();
cout << endl;
```

1g) Hva skrives ut ?

```
map<int, string> mySkis;
mySkis[100] = "Back country";
mySkis[102] = "Classic racing";
mySkis[103] = "Skate racing";
mySkis[100] = "Old Splitkein";
cout << mySkis.size() << ", " << mySkis[100];
```

1h) Hva skrives ut av denne funksjonen om den kalles slik:

```
func("aaa123");
```

```
void func(char* Cstr) {
      if (*Cstr != '\0') {
            func(Cstr + 1);
            cout << *Cstr;
      }
}
```

1i) Hva skrives ut ?

```
int found = 0, value = 5;
if (!found || --value == 0)
      cout << "danger ";
cout << "value = " << value << endl;
```

1j) Hva skrives ut ?

```
try {
      int a = 1;
      vector<int> vec;
      vec.push_back(a++);
      vec.push_back(a++);
      if (a >= 3) throw a;
      cout << vec.at(7);
}
catch (exception &e) {
      cout << "caught an exception" << endl;
}
catch (int e) {
      cout << "caught e = " << e << endl;
}
```

Oppgave 2 – Funksjoner m.m. — vindmøller (30 %)

Bakgrunn for oppgaven: Du har fått sommerjobb som programmerer i en kontroll-sentral for den store vindmølleutbyggingen på Trøndelagskysten. Du får tildelt en del mindre programmerings-oppgaver knyttet til utviklingen av en prototype på et overvåkingssystem. Hver vindmølle har en vindmåler som måler gjennomsnittlig styrke og retning for vinden det siste minuttet. Målingene samles inn av et nettverk av målere og kommunikasjonsenheter.

2a) Vindmøller må skrus av når det er alt for sterk vind, og da skal det gis en alarm for vindmøllen. Det kan også skje at en vindmåler feiler og gir ugyldige måleverdier. Implementer en funksjon

```
int checkAlarm(const double value, const double threshold)
```

Funksjonen skal returnere:

- 1 dersom **`value`** er større enn **`threshold`** og mindre eller lik konstanten **`MAX_WIND.`** Dette betyr at **`value`** er gyldig og at det skal gis en alarm.
- -1 dersom **`value`** er negativ eller større enn **`MAX_WIND`**, som betyr ugyldig måling.
- verdien 0 i alle andre tilfeller, som betyr alt OK.

2b) Implementer en funksjon

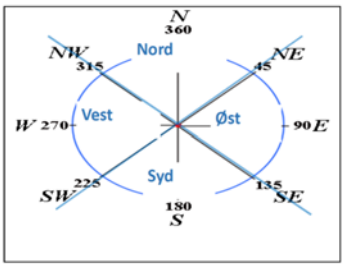
```
void printAlarm(const int id, const double wind, const int dir)
```

`id` angir vindmøllens unike nummer, **`wind`** er vindstyrken, og **`dir`** er vindens retning i grader som et heltall i området [0..359]. Funksjonen skal skrive ut alarmen på konsollet som en tekstlinje avsluttet med linjeskift. Bruk *helst* **`cout`** fra **`iostream`**. Utskriften skal være på følgende format:

```
ALARM <id>: <vindstyrke> fra <retning>
```

der **`<id>`** angir vindmøllens unike nummer. **`<vindstyrke>`** skal skrives ut med 2 desimaler etter punktum. **`<retning>`** er **`"Nord"`**, **`"Øst"`**, **`"Syd"`** eller **`"Vest"`** der 0 grader er rett mot Nord, 90 grader rett mot Øst, 180 grader rett mot Sør, og 270 er rett mot Vest. De 360 mulige verdier for grad-tallet (**`dir`**) skal avbildes til en av disse fire retningene som følger (også vist med blått i figuren til høyre):

- Øst: 45 – 134 grader
- Syd: 135 – 224 grader
- Vest: 225 – 314 grader
- Nord: 315 – 359, eller 0 – 44 grader



2c) Utbyggingen består av 278 vindmøller som er nummerert 1, 2 … 278. Målingene ankommer kontrollsentralen hvert 10. minutt i en tekstfil for alle vindmøllene. En slik fil inneholder da 10 målinger per vindmølle, totalt 2780 linjer. Hver linje beskriver en enkelt måling på formatet:

```
id vindstyrke retning
```

der `id` er vindmøllens nummer (1 … 278), `vindstyrke` er et flyttall (double) og `retning` er et helttall (grader) som beskrevet over. De første 278 linjer er målinger fra alle møllene for det 1. minuttet, deretter følger en ny sekvens av like mange målinger for 2. minutt osv. Innenfor hver blokk med 278 målinger er målingene sortert etter `id` i stigende rekkefølge 1 … 278. Vi definerer følgende datastruktur for en måling:

```
struct sample {
	int id;
	double wind;
	int dir;
};
```

id, **wind** og **dir** har samme betydning som i oppgave 2b. Overlast (Eng. overload) operatoren **<<** så det blir enkelt å skrive ut en **sample** til skjerm (konsollet):

```
ostream& operator<<(ostream& out, const sample &s);
```

Utskriften skal ha samme format som tekstfilen beskrevet ovenfor i denne deloppgaven, men skal ikke inkludere linjeskift.

2d) Vi oppretter en global tabell for å lagre alle målingene:

```
const int N_MILLS = 278;
const int N_SAMPLES = 10;
sample measurements[N_MILLS][N_SAMPLES];
```

Implementer en funksjon **void readMeasurements(const char *filename)** som leser tekstfilen med navn i **filename** og fyller ut tabellen **measurements** med målinger. Funksjonen skal åpne filen og gi en enkel feilmelding til konsollet dersom den ikke finnes, før den leser inn målingene. Du kan anta at tekstfilen er på riktig format, at alle id-verdier er korrekte og at ingen målinger mangler. (Derimot kan enkeltmålinger være ugyldige som beskrevet i oppgave 2a – dette kommer vi tilbake til i oppgave 2g).

2e) Implementer en funksjon **vector<sample> calcStats()** som regner ut statistikk for hver vindmølle, basert på informasjonen lagret i tabellen **measurements**. Funksjonen skal returnere en **vector** med én **sample** for hver vindmølle, med gjennomsnittlig vindstyrke i **sample.wind** og retningen for den sterkeste vindmålingen i **sample.dir**. Med gjennomsnittlig mener vi summen av alle verdier delt på antall verdier.

2f) Vi ønsker å sortere vektoren fra deloppgave 2e ved å bruke **sort()** fra STL som følger:

```
vector<sample> stats = calcStats();
sort(stats.begin(), stats.end());
```

For at dette skal fungere må vi overlaste mindre-enn-operatoren for **sample**:

```
bool operator<(const sample &lhs, const sample &rhs);
```

Implementer operatoren deklarert ovenfor slik at vektoren sorteres i stigende rekkefølge.

2g) Det skal *ideelt* sett være 10 målinger for hver mølle, men siden det mobile bredbåndet har liten kapasitet noen steder kan det skje at enkelte målinger mangler. Disse målingene vil få verdien -1.0 for vindstyrke i tekstfilen. Lag en funksjon **checkMeasurements**() som går igjennom målingene lagret i **measurements** fra deloppgave 2d. Funksjonen skal returnere det totale antallet målinger som mangler, og også sjekke om det mangler mer enn 1 måling for noen av vindmøllene. Straks man finner en vindmølle med mer enn 1 manglende måling skal funksjonen avbryte sjekkingen av tabellen, og kaste et unntak (Eng. throw an exception) på en slik måte at hovedprogrammet umiddelbart får beskjed og vet hvilken mølle det gjelder. (Hint: husk at man kan kaste et heltall som et unntak.)

2h) Lag en funksjon **`repairMeasurements()`** som gjør en enkel analyse av målingene i **`measurements`** fra deloppgave 2d. Funksjonen antar at det er maksimalt 1 manglende måling for hver av vindmøllene. En manglende måling skal rettes opp som illustrert ved følgende eksempel, der de 10 innleste måleverdiene vises for tre ulike møller:

```
11.1 12.2 13.3 -1.0 15.5 16.6 17.7 18.8 19.9 20.0
-1.0 12.0 13.0 14.0 15.0 15.0 14.0 18.0 10.0 12.0
11.1 12.2 13.3 14.4 15.5 16.6 15.0 15.0 14.0 -1.0
```

I den første linjen mangler vi måling nr. 4, men det kan repareres ved å sette verdien lik gjennomsnittet av verdiene på begge sider (altså 14.4). I neste linje mangler den første målingen, men den kan settes lik måleverdien på neste plass (12.0). Siste linje mangler den siste målingen, som kan settes lik verdien av forrige måling (14.0). Her er det kun måling av vindstyrke som skal rettes opp ved å gjøre de nødvendige endringene i **`measurements`**.

Oppgave 3 – NTNU-samkjøring (25 %)

Du har fått sommerjobb i den nye studentbedriften «EcoTrans» som er startet av noen miljøbevisste NTNU-studenter. Omorganiseringen til nye NTNU har gitt et behov for koordinert transport mellom byene Trondheim, Ålesund og Gjøvik. Hver uke reiser mange ansatte og studenter mellom disse byene, ofte i privatbiler med ledige seter. EcoTrans vil lage et enkelt datasystem for å stimulere miljøvennlig samkjøring, og i denne oppgaven skal vi skrive noen enkelte kodebiter for et slikt system.

3a) Deklarer en klasse **`Person`** som skal ha de private medlemsvariablene **`id`** (heltall som representerer ansatt-nr. eller student-nr.) og **`email`** (av type **`string`** som inneholder e-post-adressen). Klassen skal ha én konstruktør som setter **`id`** og **`email`** med verdier gitt av parameterlisten. Deklarer konstruktøren, de to private medlemsvariablene med aksess-funksjoner (get-funksjoner), samt en mutator-funksjon (set-funksjon) for **`email.`** (Man kan endre e-post-adresse, men ikke id).

3b) Implementer kontruktøren og get/set-funksjonene nevnt i deloppgave 3a. Bruk initialiseringsliste.

3c) Deklarer klassen **`Driver`** som er en spesialisering (subklasse) av klassen **`Person`**. Driver har to private medlemsvariabler: **`carType`** (streng, som beskriver sjåførens bil) og **`freeSeats`** (heltall, som angir antall ledige seter). Klassen har en konstruktør som sørger for at alle 4 medlemsvariablene i et **`Driver`** objekt får en verdi gitt av konstruktørens parameterliste. Implementer konstruktøren som del av klassedeklarasjonen (Eng. inline).

3d) Koden nedenfor er deklarasjon av klassene **`Participant`** og **`Meeting`**. **`Participant`** representerer en møte-deltaker. **`Meeting`** representerer et møte, på en gitt dag (**`day`**) med en start- og slutt-tid (**`start`**, **`end`**, i hele timer), og skjer i en av de tre mulig byene (**`location`**). Begge har en **`next`**-peker slik at de kan inngå i en lenket liste.

```
enum Campus { TRH, AAL, GJO };
class Participant {
private:
	Person *who;
	Participant *next;
public:
	Participant(Person *who) : who(who), next(nullptr) { }
	Person *getWho() { return who; }
	void setNext(Participant *next) { this->next = next; }
	Participant *getNext() { return next; }
};
class Meeting; // Forward declaration for Meeting
Meeting *allMeetings = nullptr; // Linked list of all Meetings
class Meeting {
private:
	int day;		// Assumes 1..365, no leap-year (ingen skudd-år)
	int start;		// Start of meeting
	int end;		// End of meeting
	Campus location;	// Where is the meeting
	Person *owner;		// Pointer to Person-object that owns the meeting
	Driver *driver;		// Points to a Driver (if there is any)
	Meeting *next; // Points to next object in linked list of Meeting-objects
	Participant *firstPart; // Points to first participant of meeting
public:
	Meeting(int day, int start, int end, Campus location, Person *owner);
	~Meeting();
	Meeting *getNext() { return next; }
	void addParticipant(Person *person);
	Participant *getParticipants() { return firstPart; }
	void printCoDriving();
};
```

I hovedprogrammet har vi en lenket liste av **`Meeting-`** objekter, og variabelen **`allMeetings`** peker til listens første element, eller **`nullptr`** ved starten av programmet. (Definisjon av variabelen er vist i koden ovenfor). Implementer konstruktøren til **`Meeting`** som oppretter et nytt møte og lenker det inn først i listen som variabelen **`allMeetings`** peker til. Husk initialisering av alle medlemsvariable. Møtets eier (`owner`) skal *ikke* legges automatisk inn som en første deltaker i deltakerlisten (`firstPart`) av konstruktøren.

3e) Implementer medlemsfunksjonen **`Meeting::addParticipant(`**`Person *person`**`)`** som legger `person` inn som møtedeltaker ved å opprette et nytt **`Participant`**-objekt for personen og legger det bakerst i listen av møtedeltakere. (Dette for å kunne tildele ledige seter i samme rekkefølge som deltakere er blitt meldt inn).

3f) Implementer funksjonen **`Meeting::printCoDriving()`** som går igjennom listen av møter (`allMeetings`) og skriver ut alle *andre* møter som har potensial for samkjøring med møte-objektet. Dette er alle møter som starter og slutter på samme tidspunkt på samme sted. Test på likhet mellom pekere for å sjekke om to møter er det samme objektet. Nedenfor er et eksempel på ønsket utskrift, med sted, dag, start, slutt og e-post til møtenes eier. Den første linjen er informasjon om møtet-objektet (**`this`**), mens de neste linjene er eierne av møtene med mulig samkjøring. Første linje skrives ut uavhengig om det finnes noen møter med mulig samkjøring. For enkel utskrift av møtested kan du anta at utskriftsoperatoren `<<` er overlagret for `Campus`. Vi skriver *ikke* ut informasjon om møtedeltakerne.

```
Possible co-driving for meeting in AAL on day 123 from 9 to 11 by <LasseN@ntnu.no>
   Meeting by <RogerM@ntnu.no>
   Meeting by <GunnarT@ntnu.no>
```

3g) Implementer destruktøren til `Meeting` som sletter alle **`Participant`** objektene som kan ha blitt opprettet med **`addParticipant()`**, slik at vi unngår minnelekkasje.

Oppgave 4 – Minnehåndtering (20 %)

I denne oppgaven skal du implementere en klasse for å behandle (1-dimensjonale) datasett av *vilkårlig størrelse*. Her defineres et datasett som en samling med N verdier $y(x_i) = y_i$ hvor x_i er et heltall (int) i området 0 … *N*-1 og y_i er et flyttall (double). Klassen skal inneholde funksjoner som gjør diverse beregninger på datasettet (her kun interpolering mellom verdier). Klassedeklarasjonen er som følger:

```
class Dataset {
private:
	double *y;
	int N;
public:
	Dataset(int N);
	Dataset(const Dataset &other);
	~Dataset();
	Dataset& operator=(Dataset rhs);
	double interpolate(double x);
	int getSize() const { return N; }
	double& operator[](int i) { return y[i]; }
};
```

4a) Implementer konstruktøren **`Dataset(int N)`**. Den skal opprette et datasett der alle verdiene er initialisert til 0.0.

4b) Implementer kopi-konstruktøren for klassen `Dataset`.

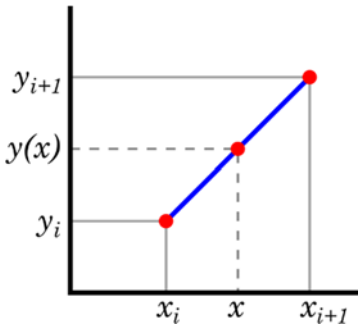
4c) Implementer destruktøren for klassen `Dataset`.

4d) Implementer tilordningsoperatoren (Eng. assignment operator). Hint: en vanlig teknikk er «copy-and-swap», men det er rom for andre løsninger.

4e) Implementer funksjonen **`double Dataset::interpolate(double x)`** som regner ut $y(x)$ ved lineær interpolering mellom punkter i datasettet. Gitt to verdier $y(x_i) = y_i$ og $y(x_{i+1}) = y_{i+1}$ i datasettet, og hvor $x_i < x < x_{i+1}$, så regnes $y(x)$ ut som følger:

$$y(x) = y_i + (y_{i+1} - y_i)\frac{(x - x_i)}{(x_{i+1} - x_i)}$$

For grenseverdier $x < 0$ og $x > N - 1$ skal funksjonen returnere henholdsvis første og siste verdien i datasettet. Utregningen er illustrert i figuren under.



4f) Implementer funksjonen **`random_data()`:**

`Dataset` **`* random_data(double min, double max, int size)`**

Funksjonen skal opprette et nytt datasett av størrelse **`size`** og fylle det med tilfeldige flyttall i intervallet **`[min, max]`**. Bruk funksjonen **`rand()`** som returnerer tilfeldige heltall i intervallet **`[0, RAND_MAX]`**.