Örebro universitet
Institutionen för teknik
Thomas Padron-McCarthy (Thomas.Padron-McCarthy@tech.oru.se)

Tentamen i Objektorienterad programmering för D2 m fl

onsdag 1 november 2006 kl 14:00 - 19:00 i L003

Hjälpmedel:	Inga hjälpmedel.
Poängkrav:	Maximal poäng är 40. För betyget 3 respektive G krävs 20 poäng.
Resultat:	Meddelas på kursens hemsida senast onsdag 22 november 2006.
Visning:	Måndag 27 november 2006 kl 12:00-12:30 i mitt rum (T2220). Efter visningen kan tentor hämtas på expeditionen.
Examinator och jourhavande:	Thomas Padron-McCarthy, telefon 070-7347013.

- Skriv tydligt och klart. Lösningar som inte går att läsa kan naturligtvis inte ge några poäng. Oklara formuleringar kommer att misstolkas.
- Skriv den personliga tentamenskoden på varje inlämnat blad. Skriv *inte* namn eller personnummer på bladen.
- Lös bara en uppgift per blad. Skriv bara på en sida av papperet. Använd inte röd skrift.
- Antaganden utöver de som står i uppgifterna måste anges. Gjorda antaganden får inte förändra den givna uppgiften.
- Skriv gärna förklaringar om hur du tänkt! Även ett svar som är fel kan ge poäng, om det finns med en förklaring som visar att huvudtankarna var rätt.

LYCKA TILL!

För alla uppgifter på tentan gäller: Man får använda både standard-C++ (som till exempel har pekare som anges med * och objekt som skapas med **new**) och Microsofts dialekt C++/CLI (som till exempel har pekare som anges med ^ och objekt som skapas med **gcnew**). Man kan dock få poängavdrag om man blandar ihop dem på ett felaktigt sätt, till exempel om man skapar ett objekt med **gcnew** och sen pekar på det med en *-pekare.

Uppgift 1 (2 p)

Vi ska testflyga en rymdraket. En viktig del av testflygningen är att samla in mätdata från olika sensorer, som till exempel mäter temperaturen på ett ställe i en raketmotor. (En "sensor" är alltså en liten apparat som mäter saker.) Helst ville vi lagra alla mätdata med hjälp av en databashanterare, men eftersom det kommer så mycket mätdata hinner databashanteraren inte med, så vi använder ett C++-program i stället.

För att lagra de uppmätta mätvärdena skapar vi klassen **Matvarde**. Specifikationen för klassen, den så kallade klassdefinitionen, ser ut så här:

```
class Matvarde {
private:
  float matvarde;
  float felgrans;
public:
  Matvarde(float matvarde, float felgrans = 0.0);
  float get_matvarde();
  float get_min();
  float get_max();
}; // Matvarde
```

Medlemsvariabeln **matvarde** anger det uppmätta mätvärdet. Det finns inga negativa mätvärden.

Tyvärr är givarna inte exakta, så det riktiga värdet kan skilja sig från vad vi mätte upp. Därför lagrar vi också en **felgrans**, som är felgränsen i procent av mätvärdet. Ett exempel är att om det uppmätta värdet är **30.0**, och felgränsen är **20.0**, så räknar man med att felet är högst **6.0**, och att det riktiga värdet alltså ligger någonstans mellan **24.0** och **36.0**. Det är de gränserna som ska returneras av funktionerna **get_min** och **get_max**.

Implementera konstruktorn och de övriga medlemsfunktionerna.

Uppgift 2 (2 p)

Vi vill testa klassen **Matvarde**. Skriv en **main**-funktion som skapar ett matvärde (dvs ett **Matvarde**-objekt) och sen kollar att de tre medlemsfunktionerna **get_matvarde**, **get_min** och **get_max** ger rätt resultat. Om alla ger rätt resultat ska programmet skriva ut **Rätt!**, annars **Fel!**.

I den här och alla andra uppgifter på tentan gäller: Om du ska använda en klass, funktion eller liknande sak från en tidigare uppgift, behöver du inte skriva koden för den på nytt. Du får också använda den även om du inte gjort uppgiften där man skulle skriva den.

Uppgift 3 (2 p)

Från varje sensor samlas en väldig mängd uppmätta värden. Vi kanske registrerar ett värde från givarna tusen gånger i sekunden. En mätning av ett värde kallar vi för en datapunkt, och för att lagra en sådan skapar vi klassen **Datapunkt**.

En **Datapunkt** ska innehålla tre privata medlemsvariabler:

- Ett mätvärde, som är ett objekt av klassen **Matvarde**.
- En tidpunkt, som lagras som ett flyttal.
- Datapunkterna ska lagras i en länkad lista, så det ska också finnas en pekare till nästa objekt i listan.

Följande medlemsfunktioner ska finnas:

- En konstruktor, som på lämpligt sätt tar emot de värden som behövs för att ge medlemsvariablerna sina värden.
- Funktionen **get_next**, som returnerar en pekare till nästa objekt i den länkade listan.
- Funktionerna **get_matvarde**, **get_min** och **get_max**, som returnerar mätvärdet respektive dess min- och maxgräns.
- Funktionen **get_tidpunkt**, som returnerar tidpunkten.

Skriv specifikationen (klassdefinitionen) för klassen **Datapunkt**.

Uppgift 4 (3 p)

Implementera konstruktorn och de övriga medlemsfunktionerna i klassen **Datapunkt**.

(Ett tips: Det finns inget krav på att datapunkterna i den länkade listan ska lagras i någon särskild ordning.)

Uppgift 5 (3 p)

Nu vill vi testa klassen **Datapunkt**. Skriv en **main**-funktion som skapar *tre* datapunkter (dvs tre **Datapunkt**-objekt), hoplänkade i en länkad lista. Sen ska funktionen gå igenom de tre objekten i listan, summera värdena från **get_min** och **get_max** för de tre objekten, och till sist skriva ut de två summorna.

Uppgift 6 (9 p)

Sensorerna i rymdraketen ska representeras av klassen **Sensor**. En **Sensor** ska lagra följande saker:

- Namnet på sensorn, till exempel **Temperaturmätare 7 i vänster motor**.
- Sensorns felgräns (i procent av mätvärdena).
- Alla mätvärdena som mätts upp sensorn. De är en länkad lista av **Datapunkt**-objekt.

Följande medlemsfunktioner ska finnas:

- En konstruktor, som tar sensorns namn och felgräns som argument.
- Funktionen **ny_matpunkt**, som anropas när vi mätt upp ett nytt värde och vill lagra det. Funktionen ska se till att skapa ett nytt **Datapunkt**-objekt, som lagras i den länkade listan. Funktionen tar bara ett argument, nämligen det uppmätta värdet, som är ett flyttal. Felgränsen för mätvärdet kan fås från sensorns felgräns. Det finns en funktion som heter **get_current_time** som kan användas för att få fram aktuell tid. Den tar inte några argument, och den returnerar tiden som ett flyttal.
- Funktionen **get_medel**, som returnerar medelvärdet av alla mätvärdena. (Medelvärdet räknas ut genom att man summerar alla de uppmätta värdena, och sen delar den summan med antalet.)

a) (2p) Skriv specifikationen (klassdefinitionen) för klassen **Sensor**.

b) (1p) Implementera konstruktorn.

c) (2p) Implementera medlemsfunktionen **ny_matpunkt**.

d) (3p) Implementera medlemsfunktionen **get_medel**.

e) (1p) Skriv en **main**-funktion som skapar en sensor, lägger in tre mätvärden med hjälp av **ny_matpunkt**, och till sist skriver ut medelvärdet som fås från **get_medel**.

Uppgift 7 (3 p)

a) (1p)

Man kan säga att varje **Sensor**-objekt "äger" en länkad lista av mätpunkter. Klassen **Sensor** har ingen destruktor. I en miljö utan skräpsamling, som standard-C++, skulle det kunna uppstå ett problem eftersom destruktorn saknas. Vilket?

b) (2p)

Implementera destruktorn.

(Om man använder en miljö som *har* skräpsamling, som C++/CLI, så behövs det egentligen inte någon destruktor, men skriv den i alla fall, som om det inte fanns någon skräpsamling.)

Uppgift 8 (3 p)

Nu vill vi testa klassen **Sensor** lite mer uttömmande. Först behöver vi en fil med testdata. Skriv ett C++-program (med **#include**-rader och allt) som skapar en fil med en miljon slumpmässiga flyttal mellan 0 och 100.

Du får själv bestämma vad filen ska heta, och om det ska vara en binärfil eller en textfil.

Tips: Funktionen **rand** ger ett slumpmässigt heltal mellan 0 och **RAND_MAX**. Med följande uttryck kan man räkna ut ett slumpmässigt flyttal mellan 0 och 100:

```
100.0 * rand() / RAND_MAX
```

Inkludera filen **cstdio** för att deklarera funktionen **rand** och konstanten **RAND_MAX**.

Uppgift 9 (4 p)

Skriv ett C++-program (med **#include**-rader och allt) som skapar en sensor, öppnar filen du skapade i uppgiften ovan, läser alla talen från filen, lägger in dem som mätvärden med hjälp av **ny_matpunkt**, och till sist skriver ut medelvärdet som fås från **get_medel**.

Du kan anta att klassdefinitionen för klassen **Sensor** finns i filen **Sensor.h**, och så vidare med eventuella andra av dina klasser som du behöver.

Uppgift 10 (9 p)

Världen består av **vassa saker** och **runda saker**. Vassa saker har en **vasshet**, som mäts med ett heltal. Runda saker har en **diameter**, som mäts med ett flyttal.

Både vassa och runda saker har en virtuell medlemsfunktion **peta**, som simulerar vad som händer om man petar på saken. Vassa sakers **peta**-funktion skriver ut **Aj!**, och runda sakers **peta**-funktion skriver ut **Rullerull!**.

Så här ser klassdefinitionerna ut:

```
class VassSak {
private:
  int vasshet;
public:
  VassSak(int vasshet);
  virtual void peta();
};

class RundSak {
private:
  double diameter;
public:
  RundSak(double diameter);
  virtual void peta();
};
```

a) (1p)

Skriv klassdefinitionen för klassen **Sax**. En sax är vass, och ska därför ärva **VassSak**. En sax har inga särskilda egenskaper förutom vassheten. Konstruktorn ska ta vassheten som argument.

b) (1p)

Implementera konstruktorn för **Sax**.

c) (1p)

Skriv klassdefinitionen för klassen **Boll**. En boll är rund, och ska därför ärva **RundSak**. Förutom diametern har en boll också en studskoefficient, som är ett flyttal. Konstruktorn ska ta både diametern och studskoefficienten som argument. När man petar på en boll ska bollen inte rulla utan studsa, så **Boll** ska ha en egen **peta**-funktion.

d) (1p)

Implementera konstruktorn för **Boll**.

e) (1p)

Implementera **peta**-funktionen för **Boll**. Den ska skriva ut **Studselistuds!**.

f) (1p)

Skapa klassen **Igelkott**. En igelkott är både rund och vass, och ska därför ärva både **RundSak** och **VassSak**. Förutom de ärvda egenskaperna har den ett **antal taggar**, som är ett heltal. Konstruktorn ska ta igelkottens alla egenskaper som argument. Igelkotten har också en egen **peta**-funktion.

g) (1p)

Implementera konstruktorn för **Igelkott**.

h) (1p)

Implementera medlemsfunktionen **peta** för **Igelkott**. Den ska först anropa **peta** i **VassSak**, för att skriva ut **Aj!**, och sen ska den anropa **peta** i **RundSak**, för att skriva ut **Rullerull!**.

Ledning till alla deluppgifterna: Med följande main-funktion:

```
int main() {
  VassSak* vs = new VassSak(10);
  RundSak* rs = new RundSak(1.8);
  Sax* s = new Sax(10);
  Boll* b = new Boll(10.0, 0.9);
  Igelkott* i = new Igelkott(17.0, 4, 10000);

  vs->peta();
  rs->peta();
  s->peta();
  b->peta();
  i->peta();
}
```

ska man få denna utmatning:

```
Aj!
Rullerull!
Aj!
Studselistuds!
Aj!
Rullerull!
```

i) (1p) Vad blir utskrifterna om man i stället har följande main-funktion?

```
int main() {
  VassSak* vs1 = new Sax(10);
  VassSak* vs2 = new Igelkott(17.0, 4, 10000);
  RundSak* rs = new Igelkott(19.3, 5, 11000);

  vs1->peta();
  vs2->peta();
  rs->peta();
}
```