Lösningsförslag till tentamen
TDP004 Objektorienterad Programmering
2008-03-26

Teoretisk del

1a). Vad är det som skiljer en virtuell funktion och en rent virtuell funktion? Du kan förutsätta att det är vanliga medlemsfunktioner som avses. (2p)
1a) En virtuell funktion deklarerar en funktion som kan överlagras. Den får implementeras i den klassen där den är deklarerad.
En rent virtuell funktion deklarerar en funktion som **måste** överlagras. Den får **inte** implementeras i den klassen där den är deklarerad (gäller C++).

1b). Vad har dessa för effekt på klassen där funktionerna är definierade? (2p)
1b) En virtuell funktion har ingen direkt effekt. Det är logiskt att virtuella funktioner används i samband med arv, men inget krav.
En rent virtuell funktion gör klassen abstrakt, dvs det går inte att skapa en instans av klassen där funktionen är deklarerad.

2. Vilka är skillnaderna mellan en `const` funktion och en `static` funktion? (2p)
En `const` funktion får inte ändra på några medlemsvariabler.
En `static` funktion delas mellan alla instanser av klassen. Den får inte använda, och därför heller inte ändra på, några klassvariabler.

3. Det finns flera centrala objekt inom objektorientering. Definiera klass och abstraktion. (4p)
Klass – datastruktur som representerar ett sammanhängande objekt med data och funktioner. Användaren kan själv definiera klasser. Klasser är den grundläggande byggstenen inom OO. Omöjligt att skapa instanser av abstrakta klasser.

Inkapsling – Separation av interface och implementation. Genom denna separation kapslar man in hur en funktion utför något och istället tillåts andra användare bara att se interfacet, som talar om vad funktionen behöver för indata och vad den returnerar. Enbart den som implementerar funktionen behöver veta exakt hur saker görs, vilket kan vara mycket komplext, de andra kan se det som en svart låda.

4. En av orsakerna till att objektorientering utvecklades var återanvändning (eng. reuse) av kod. Beskriv hur detta underlättas inom objektorientering jämfört med icke-objektorienterade språk. (2p)
Det är lättare och mera intuitivt att paketera en eller flera sammanhängande klasser jämfört med fristående funktioner i andra programmeringsparadigmer. Genom arv kan man få tillgång till basklassernas existerande funktionalitet.

5. Vad medför nyckelordet `friend` och vad har detta med inkapsling att göra? (3p)

Med hjälp av friend går man förbi de vanliga åtkomstrestriktionerna, och därigenom inkapslingen, och den klassen som deklarerar en annan klass/funktion som friend ger den tillgång till alla icke-publika data och funktioner. (Överkurs: detta anses dock inte bryta mot inkapslingen generellt eftersom det är den deklarerande klassen som ger en annan klass/funktion tillgång.)

6. I kursen pratade vi om standardbiblioteket STL. I detta ingår containrar av olika slag, tex associativa och sekventiella. Förklara likheter och skillnader mellan dem och ge exempel på en associativ och sekventiell container. (6p)

Sekventiell container:
Elementen ligger i den ordning som man lade dit dem. I fallet med vector finns omedelbar (random) access genom []-operatorn, annars måste man använda en iterator för att accessa ett visst element. Inget problem att lägga in dubletter av element. Varje element består enbart av elementet självt, inget <key, value>-pair. Exempel: std::list/ std::vector.
Associativ container:
Elementen ligger inte i samma ordning som man lade in dem, utan i annan ordning (sorterade som ett träd, eller i nycklarnas storleksordning tex). I fallet med map är varje element ett <key, value>-pair där man söker efter ett objekt med find(key). Det går snabbare att söka efter ett element i en map, men långsammare att lägga in. I en map får det inte finnas flera element med samma nyckel (men det är tillåtet i en multimap). Exempel: std::map.

Praktisk del

1. Endast funktionerna redovisas här.

```
#include <math.h>
double CalculateStatistics::CalculateMean
          (const std::list<double>& numbers) const
{
     double sum = 0.0;
     double mean;
     std::list<double>::const_iterator i;
     for(i = numbers.begin(); i != numbers.end(); ++i)
     {
          sum +=(*i);
     }
     /*Explicit conversion is not necessary but highly
     recommended, it is not performed, then an implicit
     conversion is performed. */
     mean = sum / static_cast<double>(numbers.size());
     return mean;
}

double CalculateStatistics::CalculateStandardDeviation
          (const std::list<double>& numbers) const
{
     std::list<double>::const_iterator i;
     double firstTerm = 0.0;
     double secondTerm = 0.0;
     double numerator;
     double denominator;
     double deviation;

     /*Explicit conversion is not necessary but highly
     recommended, it is not performed, then an implicit
     conversion is performed. */
     double noElements =
static_cast<double>(numbers.size());

     for(i = numbers.begin(); i != numbers.end(); ++i)
     {
          firstTerm += (*i) * (*i);

          secondTerm += (*i);
     }

     secondTerm *= secondTerm;
     secondTerm /= noElements;
```

```
      numerator = firstTerm - secondTerm;
      denominator = noElements - 1;

      deviation = sqrt(numerator/denominator);
      return deviation;
}
```

2.

```
bool Uppg2::WriteTestCaseResult(
                        const unsigned int testCaseNumber,
                        const std::string testCaseName,
                        const bool testCaseResult) const
{
      std::stringstream ss;
      ss << testCaseNumber <<"_"<<testCaseName<<".txt";
      std::string fileName = ss.str();

      /* Open file. As we use std::ofstream, we don't need
      to care about the file open modes. If we use for
      example iostream, we need to open with std::ios::out.
      */
      std::ofstream saveFile(fileName.c_str());
      if(NULL == saveFile)
      {
            return false;
      }

      saveFile << testCaseNumber
                <<" "
                <<testCaseName
                <<" "
                <<std::boolalpha<<testCaseResult
                << '\n';
      saveFile.close();
      return true;
}
```

3.

Samtliga subklassers .h-filer måste inkludera ”`FilterBaseclass.h`” och alla subklassers `.cpp-filer` måste inkludera motsvarande `.h-fil`

```
class FilterBaseclass
{
public:
      FilterBaseclass (const int aConstant);
      virtual ~FilterBaseclass (void);
      virtual int Filter (const int value) = 0;
```

```
protected:
      const int constant;

FilterBaseclass:: FilterBaseclass (const int aConstant) :
constant(aConstant)
{
}

FilterBaseclass::~FilterBaseclass (void)
{
}
```

I filen Bitcrush.h

```
class Bitcrush :
      public FilterBaseclass
{
public:
      Bitcrush(const int aConstant);
      ~Bitcrush(void);
      int Filter(const int value);
```

I filen Bitcrush.cpp

```
Bitcrush::Bitcrush(const int aConstant):
  FilterBaseclass (aConstant)
{
}

Bitcrush::~Bitcrush(void)
{
}

int Bitcrush::Filter(const int value)
{
      //No conversion should be performed.
      return ( constant *(value / constant));
}
```

I filen Tremolo.h

```
class Tremolo :
      public FilterBaseclass
{
public:
      Tremolo(const int aConstant);
      ~Tremolo(void);

      int Filter(const int value);
```

```
private:
     int n;

I filen Tremolo.cpp
#include <math.h>
Tremolo::Tremolo(const int aConstant):
  FilterBaseclass (aConstant)
{
     n = 0;
}

Tremolo::~Tremolo(void)
{
}

int Tremolo::Filter(const int value)
{
     n++;
     //Conversion to float or double is necessary.
     float change = sin(static_cast<float>(n * constant));

     //Explicit conversion is not necessary but highly
     recommended, it is not performed, then an implicit
     conversion is performed.
     return value + static_cast<int>(change);
}
```

I filen Amplitudescale.h

```
class AmplitudeScale :
     public FilterBaseclass
{
public:
     AmplitudeScale(const int aConstant);
     ~AmplitudeScale(void);

     int Filter (const int value);
};
```

I filen Amplitudescale.cpp

```
AmplitudeScale::AmplitudeScale(const int aConstant):
FilterBaseclass(aConstant)
{
}

AmplitudeScale::~AmplitudeScale(void)
{
}
```

```
int AmplitudeScale::Filter(const int value)
{
     return constant * value;
}
```