Tentamen i TDP004
Objektorienterad Programmering
Lösningsförslag

Datum:	2009-12-18
Tid:	8-12
Plats:	SU-salar i B-huset.
Jour:	Per-Magnus Olsson, tel 285607 Jourhavande kommer att besöka skrivsalarna ungefär varje timme under skrivtiden.
Hjälpmedel:	Teoretisk del: Inga. Praktisk del: Den C++ information som finns i systemet.
Betygsättning:	Max antal poäng: 44 med 22 poäng vardera på teori och praktikdel. Poäng **Betyg** 38-44 **5** 31-37 **4** 24-30 **3** 0-23 **U**
Anvisningar:	Börja med den teoretiska delen. När du är klar med den lämnar du in den och får den praktiska delen. När du har lämnat in den teoretiska delen kan du inte återvända till den. Skriv svaret på varje uppgift på ett separat blad. Uppgifterna är inte ordnade i svårighetsgrad. En kort hjälptext till Eclipse finns på sid 2-3.

Lycka till!

TDP004 Objektorienterad Programmering
Teoretisk del

1. I C++ finns tre olika typer av minneshantering.
   a) Beskriv översiktligt de olika typerna. Vilka variabler hamnar var? (3p)

   b) Var finns eventuella minnesläckor? (1p)

2. Det finns stora skillnader mellan de olika typerna av containrar i standardbiblioteket.
   a) List, vector och map är alla containrar i standardbiblioteket. De två första hör till en viss kategori och map hör till en annan kategori. Vilka är dessa kategorier? (2p)

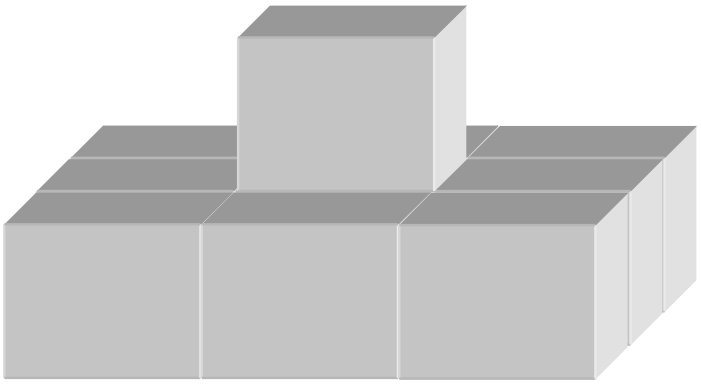
   b) Containrar i de olika kategorierna är olika lämpliga vid olika tillfällen, beskriv ett tillfälle där map är mest lämplig samt ett tillfälle där vector är mest lämplig. (2p)

3. Deklarera en klass med publik konstruktor, destruktor samt två privata funktioner varav en inte får ändra på några medlemsvariabler. Deklarera dessutom en valfri medlemsvariabel, och en medlemskonstant. För dessa två får du välja åtkomst själv. Du behöver inte implementera funktionerna, endast deklarera dem. (8p)

4. Det finns tre olika ”sorters” arv i C++. Vad är benämningarna och vad får de för konsekvenser för de funktioner och variabler som ärvs? (6p)

TDP004 Objektorienterad Programmering
Praktisk del

1. Pyramider är högsta mode (igen). Numera byggs de av identiska, rektangulära stenblock som staplas på varandra i lager. Lager 1 är alltid det översta lagret, lager 2 det näst översta osv. Bilden visar en pyramid med två lager. Lager 1 består alltid av ett enda stenblock och lager 2 består alltid av 9 stenblock, osv. Deluppgifterna kan lösas oberoende av varandra.



   A. Skriv en funktion som beräknar hur många stenblock som går åt för att skapa en pyramid med ett visst antal lager, som startar med lager 1. Givet antalet lager (vilket är ≥ 0) i pyramiden ska den returnera antalet stenblock som behövs. Svaret ska skrivas ut på skärmen tillsammans med en kort förklarande text. (4p)

   B. En del kunder tycker att det är vulgärt med toppiga pyramider och vill därför ha pyramider som inte har ett visst antal av de övre lagren. De vill t ex ha en pyramid endast bestående av lager 3-6. Skriv en funktion som beräknar antalet stenblock som behövs för en sådan pyramid. Svaret ska skrivas ut på skärmen tillsammans med en kort förklarande text. (3p)

   C. Din chef vill bygga en pyramid utanför kontoret av de överblivna stenblock som ligger och skräpar. Skriv en funktion som givet ett visst antal stenblock beräknar antalet lager i den största pyramid som kan byggas. Du kan anta att antalet stenblock är ≥ 0. Pyramiden ska börja med lager 1 och måste bestå av hela lager. Svaret ska skrivas ut på skärmen tillsammans med en kort förklarande text. (5p)

2. I katalogen given_files ligger två filer som definierar och implementerar klassen Temperaturehandler. Denna ska läsa in och spara temperaturer som läses in antingen från en fil eller genom att en funktion i klassen anropas. Klassen innehåller en mängd felaktigheter, motsägelser samt saker som inte har tänkts igenom så noggrant. Din uppgift är att hitta dessa och rätta dem på ett sätt du tycker är lämpligt. (Du kan få max 10p på den här uppgiften)

Lösningsförslag, teoretisk del

1. a) Automatiskt minne för globala variabler, static variabler et.c. Stacken innehåller stackvariabler dv.s. lokala variabler, inparametrar till funktioner. Heapen innehåller variabler allokeras med new.
b) På heapen, om man glömmer att ta bort variabler som allokeras med new så läcker det minnet, vilket man löser genom att ta bort minnet med delete.

2. a) List och vector är sekventiella containrar, medan map (tillsammans med multimap, set och multiset) är associativ.
b) Ett tillfälle då map är mest lämpligt är t.ex. då man vill associera varje element med en nyckel och gör många lookups i förhållande till antalet insert, d.v.s. ofta letar efter ett visst element. Man har då nytta av att map internt lagrar elementen i en trädstruktur vilket gör det snabbt att hitta ett visst element. Vector är mest lämpligt då man för många iterationer igenom elementen (eftersom detta är snabbt) och kan undvika att lägga till/ta bort element på andra ställen än sist.

3.

```
class Test_class
{
public:
            Test_class();
            ~ Test_class();
private:
            static void static_void_function();
            void const_function() const;
            const float float_constant;
            int int_variable;
};
```

4. Det finns publikt, skyddat och privat arv, vilket deklareras med nyckelorden public, protected samt private. Publikt arv medför ingen skillnad i åtkomst för de ärvda variablerna/funktionerna. Skyddat arv gör det ärvda till protected i subklassen medan privat arv gör det ärvda privat i subklassen.

Lösningsförslag praktisk del:
1
a)

```
int Pyramid::NoBlocksNormal(int levels)
{
      int sum = 0;               /*Total number of blocks in the pyramid so far. */
            int block_per_side;     //The number of blocks per side.
            int i;                  //Iteration variable
            if(levels == 0)
            {
                        return 0;
            }

      /* The i < levels +1 is necessary as the pyramid should contain exactly
      levels number of levels.
      For every increase in level, the number of blocks per side increases by 2. */
            for(i = 1, block_per_side = 1;
             i < levels +1;
             i++, block_per_side +=2)
            {
                        sum += block_per_side * block_per_side;
            }
            return sum;
}
```

b)

```
// Use the function from a) twice
int Pyramid::NoBlocksCapped(int upper_limit, int lower_limit)
{
```

```
            if(upper_limit > lower_limit)
            {
                        return 0;
            }
      return NoBlocksNormal(lower_limit) - NoBlocksNormal(upper_limit);
}

/* To see the diffent cases, use for example the following test cases:
no_blocks = 10 (gives 2 levels as the levels require 1+9 blocks.
no_blocks = 11 (gives 2 levels as the third level requires 25 blocks).
*/
int Pyramid::MaxLevels(int no_blocks)
{
            int sum = 0; //Total number of blocks for the pyramid so far.
            int block_per_side = 1; //The number of blocks per side.
            int level = 1;            //Iteration variable (current level)
            bool finished = false;

            if(no_blocks == 0)
            {
                        return 0;
            }

            while(finished == false)
            {
                        sum += block_per_side*block_per_side;

                        if(sum == no_blocks)
                        {
                                    //This pyramid required exactly noBlocks
blocks.
                                    finished = true;
                        }
                        else if(sum > no_blocks)
                        {
                                    /*The last level used too many blocks.
                                    Remove the last level. */
                                    level--;
                                    finished = true;
                        }
                        else
                        {
                                    //More levels can be added
                                    level++;
                                    block_per_side +=2;
                        }
            }
            return level;
}
```

2.
Följande innehåller några av de förbättringar som man kan tänkas göra. Dock är inte arbetet med att förbättra klassen klart här. Till exempel bör man implementera bättre felkontroll i de funktioner som läser indata. Det finns heller inte något sätt att läsa ut data, och inte heller att ändra gränserna för giltigt temperaturintervall.

Några av de allvarligare problemen är tex att alla medlemsvaribler och funktioner är public, samt att upper_limit inte initeras. Vidare lagras enbart data i funktionen read2 om funktionen error returnerar false, vilket den enbart gör om datan är utanför det giltiga intervallet. En funktion som kontroll om inparametern ligger inom ett giltigt intervall ska ha ett bättre namn än error, eftersom man med ledning av namnet kan tro att om funktionen returnerar falskt så har ett fel inträffat/inparametern var ogiltig.

```
#include <string>
#include <vector>

class SolutionTemperaturehandler
{
public:
            SolutionTemperaturehandler (const double a_lower_limit,
                                        const double an_upper_limit);

            //Made destructor virtual to prepare for inheritance.
            virtual ~SolutionTemperaturehandler();

            //Changed name and return value.
            bool read_from_file(std::string filename);

            //Changed parameter to be const &
            bool read_from_vector(const std::vector<double>& input);

            //Changed variables and error checking function to be protected to
prepare for use in inheritance.
protected:

            std::vector<double> temperatures;

            //Changed name and made the function const.
            bool within_limits(double x) const;

            //Changed name, made them const and removed static.
            const double lower_limit;
            const double upper_limit;
};

#include "SolutionTemperaturehandler.h"
#include <fstream>
#include <iostream>

SolutionTemperaturehandler::SolutionTemperaturehandler(const double a_lower_limit,
const double an_upper_limit) : lower_limit(a_lower_limit),
upper_limit(an_upper_limit)
{
}

SolutionTemperaturehandler::~SolutionTemperaturehandler(void)
{
}

bool SolutionTemperaturehandler::read_from_file(std::string filename)
{
            double temp;
            std::ifstream file(filename.c_str());

            if(!file)
            {
            //Added error printout and return false if failing to open file.
                        std::cout << "Temperaturehandler::read_from_file. Failed to
open file "<< filename<< std::endl;
                        return false;
            }

            while(file >> temp)
            {
                        /*Included error checking of the read data.
                        Data is only stored if within_limits returns true. */
                        if(within_limits(temp))
                        {
                                    temperatures.push_back(temp);
                        }
            }
```

```
            return true;
}

bool SolutionTemperaturehandler::read_from_vector(const std::vector<double>& input)
{
            /*Changed name of iterator variable and declared it
            on a separate line for readability. */
            std::vector<double>::const_iterator i;
            for(i = input.begin(); i != input.end(); i++)
            {
                        //Perform error checking.
                        if( within_limits(*i) )
                        {
                                    temperatures.push_back(*i);
                        }
                        /*Removed the else partto be consistent with
                         error handling in read_from_file. */
            }
            return true;
}

//Return false if the value is not within acceptable limits.
bool SolutionTemperaturehandler::within_limits(double x) const
{
            /*Strange that one limit uses less than and the other uses
            greater than or equal to, so this was changed. */
            if(x < lower_limit)
            {
                        return false;
            }

            if(x > upper_limit)
            {
                        return false;
            }

            //Changed to return true
            return true;
}
```