Lösningsförslag till tentamen
TDP004 Objektorienterad Programmering
2007-12-19

Teoretisk del

1a) Beskriv likheter och skillnader mellan en STL list och en STL vector. (4p)

Båda kan hantera olika sorters element eftersom de instantieras mha template, tex
std::vector<double> doubleVector;
std::vector<ExampleClass*> classVector;
och motsvarande för std::list.
Båda för till standardbiblioteket standard template library.

List: elementen utspridda i minnet. Snabbt att lägga till element först och sist. Långsammare iteration än vector. Minnesutrymme allokeras allt eftersom element läggs till. Egen implementation av en del algoritmer eftersom det inte finns random access-iteratorer till list.

Vector: Elementen sekventiellt efter varandra i minnnet. Snabbt att lägga till längst bak, långsamt på alla andra platser. Snabb iteration. Minnesutrymme allokeras för ett visst antal element. När detta tar slut allokeras minne för dubbla antalet element.

1b) Vilken container av STL list och STL vector är lämplig om du vet att du kommer att lägga till många element först i containern och inte kommer att iterera så många gånger genom containerns element? Motivera ditt val. (2p)

List. Snabbt att lägga till element längst fram och längst bak. Att iterationer är långsamt är inte så viktigt enligt uppgiften.

2a) Beskriv vilka tre olika sorters minneshantering det finns i C++. Vad heter de olika areorna i minnet? I vilken area hamnar static variabler, lokala variabler samt variabler som allokeras dynamiskt (6p)?

static finns i statiskt minne.
Lokala variabler och funktionsargument finns på stacken/automatiskt minne, i stack frames.
Dynamiskt allokerat minne finns på heapen/free store.

2b) Rita en översiktlig bild över de tre areorna. (1p)

2c) Inom vilken area finns eventuella minnesläckor? (1p)
Heap.

3. Det finns flera centrala objekt inom objektorientering. Definiera klass, inkapsling och arv. (6p).

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 – metod för att gömma data och implementation av funktioner i en klass, så att ingen utomstående ska vara beroende av en viss intern datatyp eller att en funktion 'är implementerad på ett visst sätt. Normalt används privat (protected) data och publika interface.

Arv – metod för att relatera olika klasser av objekt. En basklass (superklass) definierar basen för alla klasser i en arvshierarki, och subklasserna ärver data och funktionalitet från denna. Arv används ofta för att beskriva hierarkier av objekt: en katt (subklass) är ett däggdjur (basklass). I C++ finns public, protected or private arv.

4. Du har fått följande .h-fil och du ska implementera funktionerna i klassen.

```
#include <string>

struct Position
{
        double m_X;
        double m_Y;
        double m_Z;
};

class Character
{
public:
        Character(Position& aPosition, const std::string& aName, int aWeight);
        ~ Character();
private:
        Position& m_Position;
        const std::string m_Name;
        int m_Weight;
};
```

```
// m_Position och m_name måste sättas i initieringslistan.
// m_Weight kan tilldelas i konstruktorkroppen eller i initieringslistan.

Character:: Character(Position& aPosition, const std::string& aName, int aWeight) : m_Position(aPosition),
m_Name(aName),
m_Weight(aWeight)                       //Se ovan
{
        m_weight = aWeight;             //Se ovan.
}

Character::~ Character()
{
        //Inget behöver göras här.
}
```

Praktisk del

1. Skriv en funktion som tar en std::list som inparameter. Du kan förutsätta att listan endast innehåller element av typ int. Gå igenom listan framifrån. Efter att en operation har utförts på ett element ska elementet läggas till i en std::vector som benämns result. Beroende på vilket värde ett element har ska olika operationer utföras: om elementet är > 0 ska elementet kvadreras och resultatet läggas till sist i vectorn. Om elementet är 0 ska vectorn sorteras så att det minsta elementet hamnar först och sedan ska hela vectorn skrivas ut på skärmen. Elementet med värdet 0 ska inte läggas till i vectorn. Om elementet är < 0 ska funktionen returnera och inget mer ska göras. (8p)

```
#include <iostream>
#include <algorithm>
#include <list>
#include <vector>

std::vector<int> TestList(std::list<int>& aList)
{
	int value;
	std::vector<int> result;
	std::list<int>::const_iterator i = aList.begin();

	while(i != aList.end())
	{
		if((*i) > 0)
		{
			value = (*i);
			value *= value;
			result.push_back(value);
		}
		else
		{
			if((*i) == 0)
			{
				sort(result.begin(), result.end());
				PrintVector(result);
			}
			else
			{
				return result;
			}
		}
		i++;
	}
	return result;
}

void PrintVector(const std::vector<int>& aResultVector) const
{
```

```
	std::vector<int>::const_iterator i;

	std::cout<<"Vector contains the following elements: ";
	for(i = aResultVector.begin(); i != aResultVector.end(); i++)
	{
		std::cout<<(*i)<<" ";
	}
	std::cout<<std::endl;
}
```

2. Definiera en klasshierarki för att hantera olika klasser av grillar. Klassen Grill ska vara en abstrakt basklass. Alla grillar ska ha en modellbeteckning, vilket är en std::string. Modellbeteckningen anges när en instans av någon slags grill skapas. Det finns två huvudsakliga typer av grillar, *gasolgrillar* och *kolgrillar*. Alla sorters grillar ska spara bränslet som den använder samt vilket sorts grill det är.

En gasolgrill har ett visst bränsle och ett visst antal brännare. Om antalet brännare inte anges ska det ges värdet 2. Om bränslet inte anges är det ”Propan”. En gasolgrill ska kunna svara på frågan om den använder ett visst bränsle, vilket kommer i forn av en std::string. Denna funktions signatur ska vara
bool UsesFuel(const std::string& aSuggestedFuel) const.
En annan funktion som bara ska finnas för gasolgrillar ska returnera antalet brännare och dess signatur ska vara
int GetNumberOfBurners() const.

En kolgrill har alltid bränslet kol.

Följande funktioner ska finnas för alla grillar:
GetName ska returnera modellbeteckningen.
GetInformation ska returnera modellbeteckning och bränsle för en grill.
Du ska själv bestämma lämpliga returvärden och inparametrar till dessa funktioner. (10p)

Header och implementationsfiler ihopklippta här.

```
#include <string>
class Grill
{
public:
	Grill::Grill(const std::string& name):
	m_Name(name)
	{
	}

	virtual ~Grill() {};

	const std::string GetName() const
	{
		return m_Name;

	}
```

```
      virtual const std::string GetInformation() const = 0;

private:
      const std::string m_Name;
};
```

```
#include "grill.h"

class CoalGrill : public Grill
{
public:
      CoalGrill(const std::string& name, std::string& fuel =”Coal”):
      m_Fuel(fuel)
      {}
      ~CoalGrill() {}
      const std::string CoalGrill::GetInformation() const
      {
            return GetName() + “ “ + m_Fuel;
      }

private:
      const std::string m_Fuel;
};
```

```
#include "grill.h"

class GasGrill : public Grill
{
public:
      GasGrill(const std::string& name,
                        const std::string& fuel =“Propan”,
                        int burners    = 2) :
      Grill(name),
      m_Fuel(fuel),
      m_Burners(burners)
      {
      }

      ~GasGrill() {};

      bool UsesFuel(const std::string& aSuggestedFuel) const
      {
            return aSuggestedFuel == m_Fuel;
      }

      const std::string GetInformation() const
      {
            return GetName()+" "+ m_Fuel;
      }

      int GetNumberOfBurners() const
      {
```

```
            return m_Burners;
      }

private:
      const int m_Burners;
      const std::string m_Fuel;
};
```

3. Skriv om följande procedurella kod för en stackliknande datastruktur till att vara objektorienterad. Du behöver inte göra felkontroller i koden och kan anta att alla anrop till externa funktioner fungerar (6p)

Given kod utelämnad här.

```
#include <vector>

class Node;

class SpecialStack
{
public:
      SpecialStack();
      ~SpecialStack();

      bool ElementIsOnSpecialStack(const Node* aNode) const;
      bool SpecialStackIsEmpty(std::vector<Node*>& aStack) const;
      void PushSpecialStack(Node* aNode);
      Node* PopSpecialStack();
      Node* TopSpecialStack() const;

private:
      std::vector<Node*> m_Stack;
};
```

```
#include "SpecialStack.h"

SpecialStack::SpecialStack (void)
{
}

SpecialStack::~SpecialStack (void)
{
      /* Någon annan skapar Node-elementen och
       då bör denna också ta bort dem. */
}

bool SpecialStack::ElementIsOnSpecialStack(const Node* aNode) const
{
      std::vector<Node*>::const_iterator i;
      for(i = m_Stack.begin(); i != m_Stack.end(); i++)
      {
            if(aNode == (*i))
            {
```

```
                return true;
            }
        }
        return false;
}

bool SpecialStack::SpecialStackIsEmpty(std::vector<Node*>& aStack)const
{
        return (m_Stack.empty());
}

void SpecialStack::PushSpecialStack(Node* aNode)
{
        m_Stack.push_back(aNode);
}

Node* SpecialStack::PopSpecialStack()
{
        Node* node = m_Stack.back();
        m_Stack.pop_back();
        return node;
}

Node* SpecialStack::TopSpecialStack() const
{
        return m_Stack.back();
}
```