Tentamen i TDP004
Objektorienterad Programmering
Teoretisk del

Datum:	2010-12-20
Tid:	14-18
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: 40 med 20 poäng vardera på teori och praktikdel.

Poäng	**Betyg**
35-40	**5**
28-34	**4**
21-27	**3**
0-20	**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 teoretisk uppgift på ett separat blad.

Uppgifterna är inte ordnade efter svårighetsgrad.

Lycka till!

TDP004 Objektorienterad Programmering
Teoretisk del

1. Se nedanstående klass

```
class Entity {

public:

   Entity();

   ~ Entity();

   /*Updates the position etc, deltaTime is the time

   since the last update*/

   void Update(double delta_time)

protected:

   int m_id;

   double m_speed

   double m_x;

   double m_y;

   double m_z;

 private:

   void Fight(Entity* opponent = m_old_opponent);

   Entity* m_old_opponent

};
```

Antag att klassen `Troll` ärver från `Entity`.

   a) Vilken tillgänglighet får funktionen `Update` och variabeln `m_id` om arvet är `public`? (2p)

   b) Vilken tillgänglighet får funktionen `Fight` och variabeln `m_id` om arvet är `private`? (2p)

2. I standardbiblioteket finns en mängd olika containrar av olika typer. Många,, men inte alla, funktioner finns för flera containrar. Till exempel finns funktionen `push_back` för både `vector` och `list`, medan `push_front` finns till `list` men inte till `vector`. Motivera varför. För full poäng krävs en generell motivering som dels nämner den del av designfilosofin bakom STL som är relevant här, och dels ett exempel på varför `push_front` finns till `list`, men inte till `vector`. (3p)

3. Funktioner och variabler som är static är lite speciella. Beskriv vad som skiljer en static funktion från en vanlig klassfunktion samt vad som skiljer en static variabel från en vanlig lokal/ klass-variabel. (5p)

4. I C++ finns något som kallas namespace.

    a. Vad är fördelen med namespace? (2p)

    b. På vilka två sätt kan man göra för att använda en funktion som finns i ett visst namespace? (2p)

5. Antag att vi har en typ `T`. Ett objekt av typen `T` ska vara inparameter till en funktion, i vilken objektet inte får ändras. Hur skulle du göra parameteröverföringen i följande fall. Motivera!

    a. Om `T` är grundläggande typ, till exempel `int`? (2p)

    b. Om `T` är en mera komplicerad typ, t.ex. en klass? (2p)

Lösningsförslag

Teoretisk del

1.

    a. Båda behåller den tillgänglighet de har i `Entity`. `Update` är `public` och `m_id` är `protected`.

    b. `Fight` kan inte ärvas eftersom den är `private` och `m_id` blir `private` i `Troll`.

2. En av designfilosofierna för STL är att enbart tillhandahålla funktioner som är effektiva, d.v.s. som tar kort tid att exekvera. `push_back` är effektiv eftersom inte behöver flytta på några element som redan finns i containern. `push_front` är effektiv i en `list` eftersom endast några pekare behöver ändras. I en `vector` skulle alla element behöva flyttas, eftersom elementen är garanterade att ligga sekventiellt efter varandra, i ett sammanhängande minnesutrymme. Eftersom detta inte kan göras på ett effektivt sätt så har man bestämt att `push_front` inte ska finnas som funktion i `vector`.

3. För medlemsfunktioner och medlemsvariabler:

   Gemensam för alla instanser av klassen. Kan användas även om det inte finns någon instans av klassen.

   En static funktion får enbart anropa andra funktioner som är static eller const. Kan inte använda medlemsvariabler (om de inte kommer som inparametrar).

   En static medlemsvariabel måste initieras separat, utanför alla funktionerna i klassen.

   För en lokal variabel som är static: Variabeln och dess värde kvarstår mellan anropen. Minnet allokeras enbart en gång.

4.

    a. Fördelen med ett namespace är att man kan återanvända namn på funktioner och variabler så länge de ligger i olika namespace. namespace ger också en naturlig uppdelning av program, där de olika delarna finns i olika namespace.

    b. Man kan dels skriva `using namespace` namn och därigenom få tillgång till allt som finns i `namespace` namn, och dels kan man skriva namespacenamn::funktionsnamn, t.ex. `std::max` och därigenom anropa funktionen max i som finns i namespace `std`.

5.

    a. I det här fallet man kan faktiskt använda både ”call by name” och ”call by reference”, d.v.s. både `f(const int a)` och `f(const int& a)` om funktionen heter `f`. Eftersom typen inte tar så stor plats i minnet så tar det inte något lång tid att kopiera data och det finns därför inget stort behov av att använda referenser, men det är inte direkt fel att göra det.

b. I det här fallet bör man använda ”call by reference”, `f(const complicated_type& a)` för att undvika onödig kopiering av data. Man kan även tänka sig att använda `f(complicated_type* const a)` om man vill använda pekare, och se till att objektet inte kan ändras i funktionen. Man naturligtvis även göra pekaren `const`.

Lösningsförslag praktisk del

1.

```
#include <iostream>
#include <string>

class Food
{
public:
	Food(float weight): m_weight(weight), m_seconds_per_hour(3600),
joule_per_watt_hour(3600) {};

	virtual ~Food(void) {};

	virtual double CalculateEnergy()
	{
		return 0;
	}
protected:
	double m_weight;
	double m_seconds_per_hour;
	double joule_per_watt_hour;

};

class Potatoes: public Food
{
public:
	Potatoes(float weight): Food(weight)
	{
		m_energy_required = 6.5;
	};

	~Potatoes() {};

	double CalculateEnergy()
	{
		double total_energy = m_energy_required *
m_seconds_per_hour/joule_per_watt_hour;
		return total_energy;
	}
protected:
	double m_energy_required;
};


class Ham: public Food
{
public:
	Ham(float weight): Food(weight) {};

	~Ham() {};
};

class Meatballs: public Food
```

```
{
public:
	Meatballs(float weight): Food(weight)
	{
		m_energy_required = 10.0;
	};

	~Meatballs() {};

	double CalculateEnergy()
	{
		double total_energy = m_weight * m_energy_required *
m_seconds_per_hour / joule_per_watt_hour;
		return total_energy;
	}

protected:
	double m_energy_required;
};

int main(int argc, char* argv[])

{

	Food* ham = new Ham(12.0);
	Food* pot = new Potatoes(3.0);
	Food* meatballs = new Meatballs(4.0);

	double ham_energy = ham->CalculateEnergy();
	double pot_energy = pot->CalculateEnergy();
	double meatballs_energy = meatballs-> CalculateEnergy();

	cerr<<"Energy for ham "<< ham_energy << " Wh" << endl;

	cerr<<"Energy for potatoes "<< pot_energy<< " Wh" << endl;

	cerr<<"Energy for meatballs "<< meatballs_energy<< " Wh" << endl;

	double total_energy = ham_energy + pot_energy + meatballs_energy;

	cerr << "Total energy required for all food "
		<< total_energy << " Wh" << endl;
	delete ham;
	delete pot;
	delete meatballs;

	return 0;
}
```

2.

```
#include <string>
#include <iostream>

using namespace std;

class Groensfeld
{
public:
    Groensfeld (void) : first_uppercase_character(65), first_digit(48)
    {
        alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
        alphabet_length = static_cast<int>(alphabet.size());
        cerr << "Length is "<< alphabet.size() << endl;
    }
    ~ Groensfeld (void) {};

    int get_index(char a)
    {
        return static_cast<int>(a);
    }

    char get_letter(int position)
    {
        if(position >= alphabet_length)
        {
            position -= alphabet_length;
        }
        if(position < 0)
        {
            position += alphabet_length;
        }

        return alphabet[position];
    }

    // Returns the number of positions this letter shall be shifted.
    int get_shift(const string& key)
    {
        int shift = key[current_index];
        shift -= first_digit;
        current_index++;
        current_index = current_index % key.size();
        return shift;
    }

    //Cipher the string in, using the key.
    string cipher(const string& in, const string&  key)
    {
        current_index = 0;
        string outString;
        unsigned int i;
        int index ;
        int shift;

        for(i = 0; i < in.size(); i++)
```

```
        {
            if(in[i] != ' ')
                  {
                        index = get_index( in[i]);
                        index -= first_uppercase_character;
                        shift = get_shift(key);
                        index += shift;
                        outString.push_back( get_letter(index) );
                  }
        }
        return outString;
    }

    //Decipher the string in, using the key.
    string decipher(const string& in, const string& key)
    {
        current_index = 0;
        string outString;
        unsigned int i;
        int index ;
        int shift;

        for(i = 0; i < in.size(); i++)
        {
                  if(in[i] != ' ')
                  {
                        index = get_index( in[i]);
                        index -= first_uppercase_character;
                        shift = get_shift(key);
                        index -= shift;
                        outString.push_back( get_letter(index) );
                  }
            }
        return outString;
    }

     string alphabet;
     int alphabet_length;           // Length of alphabet
     int current_index;             // Current index in key.
     const int first_uppercase_character; // ASCII code for first upper
case character
     const int first_digit;         // ASCII code for first digit.
};


int main(int argc, char* argv[])

{

      string key;
      string mess;
      cerr <<"Ange nyckel: "<< endl;
      cin >> key;
      fflush(stdin);

      cerr << "Ange text: "<< endl;
      getline(cin, mess,'\n');
```

```
        GroensfeldCipher c = GroensfeldCipher();
        string ciphermess = c.cipher_in(mess, key);
        cerr<<"Chiffrerat meddelande: "<< ciphermess << endl;
        string deciphermess = c.decipher(ciphermess, key);
        cerr<<"Dechiffrerat meddelande: "<< deciphermess << endl;

        return 0;

}
```