Concordia University

Department of Electrical and Computer Engineering

FINAL EXAMINATION
(FALL 2010)

COEN 244 – Programming Methodologies II

Department of Electrical and Computer Engineering
Concordia University

December 16, 2010
Duration: 3 hours

Instructions:

- Closed book and no calculators allowed.
- Clearly state all assumptions.
- Handwriting must be legible.
- The quality of your code will also be evaluated
- Questions sheet must be signed and handed in
- All answers must be written in the answering book and not on the question sheet

I am aware of the university regulations concerning cheating and plagiarism: [ ] yes, [ ] no.

Consistent with the code of ethics of my (future) profession,
I will not cheat during this examination:

_____	_____	_____
Student ID	Printed Name	Signature

Part I: University Registrar System [65 marks]

You are in charge of developing a "University Registrar System" for managing course offerings and students. Each student (class `Student`) has three member variables: (1) basic student data (class `BasicStudentData`), (2) an array (of fixed size `maxNumberOfCourseGrades`) of grades (class `CourseGrade`) that the student has obtained so far for all successfully completed courses and (3) a variable (`int`) that stores the number of grades obtained so far. Each grade (class `CourseGrade`) is associated with a description of the course (class `CourseDescription`) in which the grade was obtained and the mark (`double`) obtained for that course. The class `CourseDescription` is defined by a course code (`int`), a course name (`string`) and the number of credits (`int`) given for the course.

A course offering (class `CourseOffering`) represents a course offered at the university. It is defined by a course description (class `CourseDescription`), the year (`int`) the course was/is offered, a capacity (`int`) indicating the maximum number of students that can be registered to the course and the number of students (`int`) actually registered with the course. Additionally, each course offering contains an array of basic student data (class `BasicStudentData`) for each student registered to the course.

A central Registrar (class `Registrar`) is used to manage all the students (class `Student`) in the system as well as the various course offerings (class `CourseOffering`).

The definitions (header files) of classes `BasicStudentData`, `CourseGrade` and `CourseDescription` are given below:

```
class BasicStudentData {
public:
  BasicStudentData();
  BasicStudentData(int studentId, string name);

  // Getters and Setters
  int getId() const;
  string getName() const;
  void setId(int studentId);
  void setName(string name);

  // Prints the id and name of a student
  void printStudentData()const;

private:
  int studentId;
  string name;
};

#include "CourseDescription.h"
class CourseGrade {
public:
  CourseGrade();
  CourseGrade(CourseDescription desc, double mark);

  double getMark() const;
  CourseDescription getCourseDescription() const;
  void setMark(double mark);
  void setCourseDescription(CourseDescription& desc);

private:
  CourseDescription courseDesc;
  double mark;
};
```

```
class CourseDescription {
public:
  CourseDescription(int courseCode=0, string courseName="", int credits=0);
  int getCourseCode() const;
  string getCourseName() const;
  int getCredits() const;
  void setCourseCode(int courseCode);
  void setCourseName(string courseName);
  void setCredits(int credits);

private:
    int courseCode;
    string courseName;
    int credits;
};
```

We also note that class `Exception` (will be used later on) already exists and has the following definition:

```
class Exception {
public:
   Exception(string message);
   string what() const;
private:
   string message;
};
```

Question 1 [15 marks]
Consider the definition (header file) of class `Student`

```
//Class Student
#include <string>
#include <iostream>
using namespace std;

#include "CourseGrade.h"
#include "BasicStudentData.h"

class Student {
public:
  Student();
  Student(int id, string name);
  Student(const Student& rhs);
  virtual ~Student();

  void addCourseGrade(const CourseGrade& grade);
  void removeCourseGrade(int courseCode);
  double getAverageGrade() const;
  void listAllGrades() const;

  // Getters and Setters
  void setBasicStudentData(BasicStudentData data);
  BasicStudentData getBasicStudentData() const;
  int getNumberOfCourses() const;

private:
  BasicStudentData data;
  static const int maxNumberOfCourseGrades = 100;
  CourseGrade grades[maxNumberOfCourseGrades];
  int numberOfGrades;
};
```

a) Provide the implementation for the default constructor: [2 marks]

```
Student::Student()
```

The constructor shall properly initialize the members `data` and `numberOfGrades` with appropriate default values.

Solution:

```
Student::Student() {
  data = BasicStudentData();
  numberOfGrades = 0;
}
```

b) Provide the implementation for the constructor: [3 marks]

```
Student::Student(int id, string name)
```

The constructor shall properly initialize the members `data` and `numberOfGrades`.

Solution:

```
Student::Student(int studentId, string name) {
  data = BasicStudentData(studentId, name);
  numberOfGrades = 0;
}
```

c) Provide the implementation for:

`void Student::listAllGrades() const` [5 marks]

The function prints out the basic student data (`id`, `name`) and a list of all the course grades obtained by the student. For each grade, the name of the course as well as the mark obtained should be printed. Assume that grades range from [`0..numberOfGrades`].

Solution:

```
void Student::listAllGrades() const {
  for (int i = 0; i < numberOfGrades; ++i) {
    data.printStudentData();
    cout << grades[i].getCourseDescription().getCourseName() << ": "
         << grades[i].getMark() << endl;
  }
}
```

d) Provide the implementation for [5 marks]:

`double Student::getAverageGrade() const`

The function calculates the average of the all grades obtained by the student.

Solution:

```
double Student::getAverageGrade() const
{
  if (numberOfGrades == 0) return 0.0;

  double total = 0.0;
  for (int i = 0; i < numberOfGrades; ++i)
      total += grades[i].getMark();
  return total / numberOfGrades;
}
```

Question 2 [30 marks]
Consider the definition (header file) of class `CourseOffering`

```
//Class CourseOffering
#include "BasicStudentData.h"
#include "CourseDescription.h"

class CourseOffering {
public:
   CourseOffering(int code, string name, int credits, int year, int capacity);
   CourseOffering(const CourseOffering& rhs);
   virtual ~CourseOffering();

   void addStudentData(BasicStudentData& data);
   void removeStudentData (int id);

   //Getters and Setters
   CourseDescription getCourseDescription() const;
   void setCourseDescription(CourseDescription info);
   void setYear(int year);
   int getYear() const;
   int getNumberOfStudents() const;
   BasicStudentData* getRegisteredStudents() const;

private:
   CourseDescription desc;
   int year;
   int capacity;
   int numberOfStudents;
   BasicStudentData* registeredStudents;
};
```

a) Provide the implementation for the constructor [6 marks]:

```
CourseOffering::CourseOffering(int code, string name, int credits,
                               int year, int capacity)
```

The constructor shall properly initialize the data members `desc`, `year`, `capacity` and `numberOfStudents`. Additionally it should *dynamically* create the array `registeredStudents` of size `capacity` and initialize it with default values. Shall the `capacity` be less or equal to zero an exception (object of class `Exception`) with an appropriate error message shall be thrown.

Solution:

```
CourseOffering::CourseOffering(int code, string name, int credits,
                               int year, int capacity)
{
  this->desc = CourseDescription(code, name, credits);
  this->year = year;
  this->capacity = capacity;
  this->numberOfStudents = 0;

  if (capacity > 0) {
       registeredStudents = new BasicStudentData[capacity];
       for (int i = 0; i < capacity; ++i)
              registeredStudents[i] = BasicStudentData();
       }
       else throw Exception("Capacity must be greater than 0");
}
```

b) Provide the implementation for the copy constructor.

`CourseOffering::CourseOffering(const CourseOffering& rhs);` [6 marks]

Solution:

```
CourseOffering::CourseOffering(const CourseOffering& rhs) {
  desc = rhs.desc;
  year = rhs.year;
  capacity = rhs.capacity;
  numberOfStudents = rhs.numberOfStudents;

  //copy the array;
  registeredStudents = new BasicStudentData[capacity];
  for (int i = 0; i < capacity; ++i)
      registeredStudents[i] = rhs.registeredStudents[i];
}
```

c) Provide the implementation for the destructor.

`CourseOffering::~CourseOffering()` [2 marks]

Solution:

```
delete [] registeredStudents;
```

d) Provide the implementation for:

`void CourseOffering::addStudentData(BasicStudentData& data)` [8 marks]

This function adds a student data to the end of the array `registeredStudents` and increments `numberOfStudents`. The data should only be added, if the student is not already in the list *and* if the course offering is not yet at capacity. Throw an exception otherwise.

Solution:

```
void CourseOffering::addStudentData(BasicStudentData data)
{
   if (numberOfStudents == capacity) throw Exception("Course Offering is Full");
   registeredStudents[numberOfStudents] = data;
   numberOfStudents++;
}
```

e) Provide the implementation for:

`void CourseOffering::removeStudentData(int id)` [8 marks]

This function decrements `numberOfStudents` and removes the data record of the student with corresponding `id` from `registeredStudents` by shifting the array from right to left. If the student does not exist, an exception with an appropriate error message shall be thrown.

Solution:

```
int CourseOffering::findStudent(int aStudentId) const
{
  int result = -1;
  for (int i = 0; i < numberOfStudents; ++i)
       if (registeredStudents[i].getId() == aStudentId) {
              result = i;
              break;
       }
return result;
}

void CourseOffering::removeStudentData(int aStudentId)
{
   int position = findStudent(aStudentId);
   if (position == -1) throw Exception("Student is not in Course Offering");

   for (int i = position; i < numberOfStudents-1; ++i) {
       registeredStudents[i] = registeredStudents[i+1];
   }
   registeredStudents[numberOfStudents-1] = BasicStudentData();
   numberOfStudents--;
}
```

Question 3 [10 marks]

The `Registrar` class below represents information about a registrar. It uses two arrays (`students` and `courseOfferings`) for storing all registered students and all course offerings provided by the university. The current number of students and course offerings in the system is given by the member variables `numberOfStudents` and `numberOfCourseOfferings`, respectively.

It also contains basic query and backup member functions.

```
//Class Registrar
#include "Student.h"
#include "CourseOffering.h"

class Registrar {
public:
  Registrar();
  void addStudent(Student& s);
  void addCourseOffering(CourseOffering& c);
  void listCourseOfferings (Student& s) const;
  void backup(int year) const;
private:
  Student* students;
  CourseOffering* courseOfferings;

  int numberOfStudents;
  int numberOfCourseOfferings;
};
```

a) Provide the implementation for [10 marks]:

```
void Registrar::listCourseOfferings (Student& s) const
```

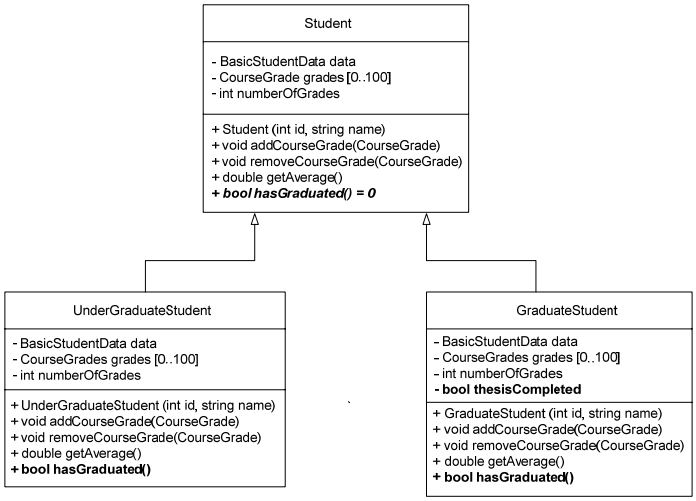
This function determines the course offerings the student is *currently* (year 2010) registered with. For each found course offering, the course name shall be printed.

Solution:

```
void Registrar::listCourseOfferings (Student& s) const {
 for (int i = 0; i < numberOfCourseOfferings; ++i) {
  if (courseOfferings[i].getYear() == 2010) {
    BasicStudentData* registeredStudents = courseOfferings[i].getRegisteredStudents();
    for (int j = 0; j < courseOfferings[i].getNumberOfStudents(); ++j) {
        if (registeredStudents[j].getId() == s.getBasicStudentData().getId()) {
           cout << courseOfferings[i].getCourseDescription().getCourseName() << endl;
        }
    }
  }
 }
}
```

Question 4 [10 marks]:

We want to improve the "University Registrar System" by distinguishing between two types of students: undergraduate and graduate students. The main difference between the two in the context of this exam is that undergraduate students need to complete 145 credits to graduate (i.e. finish their studies), whereas graduate students need to complete 12 credits and complete a thesis. To accommodate this new requirement, we have decided to add the pure `virtual boolean` member function `hasGraduated()` to class `Student`. We also introduce the two classes `GraduateStudent` and `UnderGraduateStudent` that derive from `Student` and override `hasGraduated()`.



*Getters and Setters are omitted in the diagram.

a) Provide the implementation for constructor of class `GraduateStudent` [3 marks]

```
GraduateStudent::GraduateStudent(int id, string name)
```

Solution:
[Note that the diagram contained a typo. The member variable 'data' in both derived classes was of type BasicData and not BasicStudentData]

```
void GraduateStudent::GraduateStudent(int id, string name) :
     Student(id, name), thesisCompleted(false);
```

The constructor shall properly initialize the data members of the class.

b) Provide the implementation for:

```
bool GraduateStudent::hasGraduated() const
```
[7 marks]

The function shall be implemented according to the above mentioned graduation criteria.

Solution:

```
bool GraduateStudent::hasGraduated() const {
int numberOfCredits = 0;
for (int i = 0; i < numberOfGrades; ++i)
  numberOfCredits += grades[i]. getCourseDescription().getCredits();

if ((numberOfCredits >= 12) && thesisCompleted) return true
else return false;
```

Part II: Operator Overloading [15 marks]

A three dimensional vector in mathematics is defined as follows: $v = ai + bj + ck$, where *a*, *b*, *c* are real numbers and *i*, *j*, *k* are unit vectors along x, y and z-axis respectively.

Given the following two vectors,

$v_1 = a_1 i + b_1 j + c_1 k$

and

$v_2 = a_2 i + b_2 j + c_2 k$,

The *dot* product of two vectors is scalar and is defined as follows:

$$v_1 \cdot v_2 = a_1 a_2 + b_1 b_2 + c_1 c_2$$

The *cross* product of two vectors is defined as follows:

$$v_1 * v_2 = (b_1 c_2 - c_1 b_2) i + (c_1 a_2 - a_1 c_2) j + (a_1 b_2 - b_1 a_2) k$$

Consider the definition of class `Vector` given below:

```
#include <iostream>
using namespace std;
class Vector {
friend ostream& operator<< (ostream& os, const Vector& v);

public:
    Vector(double xcomponent, double ycomponent, double zcomponent);

    double operator+ (const Vector& v) const;
    Vector operator* (const Vector& v) const;

private:
    double xcomponent;
    double ycomponent;
    double zcomponent;
};
```

a) Overload the (+) operator to perform *dot product* of two vectors [4 marks]:

```
    double Vector::operator+ (const Vector& v) const;
```

Solution:

```
double Vector::operator+(const Vector & v) const
{
  return xcomponent * v.xcomponent +
         ycomponent * v.ycomponent +
         zcomponent * v.zcomponent;
}
```

b) Overload the (*) operator to perform *cross product* of two vectors [7 marks]:

```
    Vector Vector::operator* (const Vector& v) const;
```

Solution:

```
Vector Vector::operator*(const Vector & v) const
{
  double x = (ycomponent * v.zcomponent) - (zcomponent * v.ycomponent);
  double y = (zcomponent * v.xcomponent) - (xcomponent * v.zcomponent);
  double z = (xcomponent * v.ycomponent) - (ycomponent * v.xcomponent);
```

```
    return Vector(x,y,z);
}
```

c) Provide an implementation for [4 marks]:

ostream& operator<< (ostream& out, const Vector& v)

This function prints outs the components of a vector in the following format: $ai + bj + ck$ (e.g., $3.1i + 4.6j + 0.3k$)

Solution:

```
ostream& operator<< (ostream& os, const Vector& v) {
  os << v.xcomponent << "i + " << v.ycomponent << "j + " << v.zcomponent << "k";
  return os;
}
```

Part III: Problem Analysis [20 marks]

Question 1 [10 marks]:

Your manager is unhappy with the performance of the **`listCourseOfferings(Student&)`** function of the `Registra` class. "The algorithm seems awfully complex for such a basic function!" You are tasked to redesign the system such that the performance of this function will be improved. Your answer should relate to the following problem analysis steps:

1. **Problem identification:** What is the essence of the problem we are trying to solve? (2 marks)
2. **Problem modeling:** Which variables are needed? Do we need new classes, new class compositions, inheritance, etc.? (2 marks)
3. **Problem analysis:** Think of different alternatives, options. Discuss the pros and cons. (2 marks)
4. **Problem solution:** Which solution fits best and why? (2 marks)

Be brief and concise. You do *not* need to implement the proposed solution

Question 2 [10 marks]:

Recall the *redesign* of the "University Registrar System" (Part I, Question 4).

a) What is the advantage of creating two derived classes to represent the two types of students?

b) Is there another alternative solution? Discuss.