Instructions: (Read carefully before the exam begins):

1. Before you begin, make certain that you have one Exam Booklet with pages numbered 1–12 printed double-sided.

2. There are a total of 6 questions, for a total of 90 marks.

3. The marks assigned to each question are shown at the beginning of the question; use this information to organize your time effectively.

4. Place all your answers in the spaces provided on these pages. There is 1 extra sheet at the end of the exam that can be used to continue the answer of any question.

5. You do not need to write comments in your code unless it is specifically required by the question.

6. All C++ questions should be answered using C++03 (not C++11).

7. Questions will not be interpreted. Proctors will only confirm or deny errors in the questions. If you consider the wording of a question to be ambiguous, state your assumptions clearly and proceed to answer the question to the best of your ability. You may not trivialize the problem in your assumptions.

8. Cheating is an academic offence. Your signature on this exam indicates that you understand and agree to the University’s policies regarding cheating on exams.

Signature: _________________________________________________
**Question 1 [25 marks] ADT Design**

Design an ADT for valid email addresses. An email address consists of

- a userid, which starts with an alphabetic character and is followed by one or more alphanumeric characters
- a hostname made up of one or more names separated by '.', where each name starts with an alphabetic character and is followed by one or more alphanumeric characters

For simplicity, assume that userids cannot include '.', and that neither userids nor hostnames can include special characters like '_'. Any value outside of the format described above is considered an invalid email address.

Your answer to this question is in several parts. **Read all of the parts of the question before starting your answer.**

a) Provide a complete header (EAddr.h) for your ADT, including the full class definition; declarations of nonmember functions, operations, exceptions; and #include directives and header guard. Place all of your ADT definitions in the same namespace, named EAddr. Define a useful public interface, including overloaded operations. Use default arguments to simplify the ADT interface, and use `const` to indicate when a function modifies its parameters or its object. (Do not provide implementations of member or nonmember functions.)

b) Your ADT should include an exception class to be used whenever a function attempts to produce an email address that lies outside of the valid format. The name of the exception class should be `InvalidFormat`. The exception class should store the attempted invalid value and should provide a member function for retrieving this value.

c) State whether your ADT is mutable or immutable. Defend your design decision. Explain (briefly) what implementation decisions are needed to make your ADT mutable or immutable.

d) For each of the following, explain briefly why your class definition explicitly declares, explicitly prohibits, or uses the compiler-generated version of the function:

- default constructor
- destructor
- copy constructor
- assignment operator
Questions 2–4 refer to the following code.

- Class `Workday` records information about an Employee's work performance on a particular day (hours worked, amount of sales for the day).
- Class `Employee` maintains information about an employee, including name, hourly pay rate, and work-performance records. It also provides functions for adding a new work-performance record, computing the salary earned during a time period; and a protected function for retrieving a copy of the work-performance records for a time period.

```cpp
class Day {
public:
    Day (int, int, int); // returns the year
    int year(); // returns the month (1–12)
    int month(); // returns the day of the month (1–13)
    int day();
    bool operator== (Day); // are the dates of two days equal?
    bool operator!= (Day); // are the dates of two days unequal?
    bool operator< (Day); // is one day earlier than another?
    bool operator<= (Day); // is one day earlier or the same as another?
    bool operator> (Day); // is one day later than another?
    bool operator>= (Day); // is one day the same or later than another?
private:
    int year_; // returns the number of hours worked that day
    int month_; // returns the amount of sales made that day (in $ and ¢)
    int day_; };

class Workday : public Day {
public:
    Workday(int day, int month, int year, int hours, float sales);
    int hours(); // returns the number of hours worked that day
    float sales(); // returns the amount of sales made that day (in $ and ¢)
private:
    int hours_; // returns the number of hours worked that day
    float sales_; };

class Employee {
public:
    Employee(string, float); // returns name of employee
    virtual string name(); // returns hourly rate of employee
    virtual float hourly_rate(); // returns the number of hours worked that day
    virtual float pay(Day d1, Day d2); // wages earned in day range [d1..d2], inclusive
    virtual void work(Workday*); // inserts record into records_ (sorted by Day)
protected:
    // returns all work records for day range [d1..d2], inclusive
    vector<Workday*> range(Day, Day);
private:
    string name_; // returns the number of hours worked that day
    float hrate_; // returns the amount of sales made that day (in $ and ¢)
    vector<Workday*> records_; // employee's work records
};
```
Question 2 [10 marks] Object-Oriented Programming

You are to define and implement a C++ class ComEmp derived from the provided Employee class, so that ComEmp employees earn commissions on the sales they make (i.e., a percentage of the amount of the sales) in addition to earning an hourly wage. Suppose Sara is a ComEmp who earns a 6% commission on sales and earns $20/hr in wages. If in the course of a week Sara works 20 hours and makes $5000 in sales, then her pay for that week is ($400+$300 = $700).

To get full marks on this question, your ComEmp must

1. Behave like an Employee with the exception that ComEmp employees earn a commission on the sales that they make.

2. Reuse code from the Employee base class as much as possible.
Question 3  [14 marks] - STL

You are to implement two of the member functions of the Employee class (defined on page 6) using STL containers and algorithms, functors, and functor adapters.

a) Implement Employee::range(Day start, Day end) to find the subrange of entries in Employee::records_ whose dates fall in the day interval [start..end], inclusive. You may assume that Employee::records_ is already sorted in increasing order according to the records' Day value.

b) Implement Employee::pay (Day start, Day end) to compute the wages that the employee has earned based on the employee's hourly rate and the number of hours that the employee has worked during the day interval [start..end], inclusive.

Your program must include all necessary libraries and must declare and define any function or class that it uses.

vector<Workday*> Employee::range(Day start, Day end) {

float Employee::pay( Day start, Day end ) {

a) (9 marks) Favour Composition over Inheritance

In question 2, Employee and ComEmp are related by inheritance, but ComEmp functionality could have been implemented instead using object composition and delegation.

i. Provide a complete UML model of this alternative design, including all attributes and operations of all classes, operation signatures, appropriate association types (e.g., aggregation, composition), and multiplicities.

ii. Explain which of these two designs of ComEmp (inheritance versus object composition with delegation) is better and why.

![UML Diagram]

- **Day**
  - day
  - month
  - year
  + Day (int, int, int)
  + day() : int
  + month() : int
  + year() : int

- **Workday**
  - hours
  - sales
  + Workday(int, int, int, int, float)
  + hours() : int
  + sales() : int

- **Employee**
  - name
  - hrate
  + Employee(string, float)
  + name() : int
  + hourly_rate() : float
  + pay(Day, Day) : float
  + work(Workday*) : void
  + range(Day, Day) : vector<Workday*>
b) (6 marks) Open/Closed Principle
Does your redesign of an Employee / ComEmp in question 4a adhere to the Open/Closed design principle? If not, then use the Dependency Inversion Principle to provide below a modified version of your answer to question 4a, such that your new design does adhere to the Open/Closed principle. Explain how your new design (or your answer to question 4a) is both "open" and "closed".

```
Employee
- name
- hrate

+ Employee (string, float)
+ name() : int
+ hourly_rate() : float
+ pay(Day, Day) : float
+ work(Workday*) : void
+ range(Day, Day) : vector<Workday*>
```
Question 5 [11 marks] Design Patterns

Below is a partial UML model of the Employee and Workday classes defined and implemented in questions 2 and 3. In this question, you are to consider Commission as a form of decorator that can augment an Employee object so that the employee can receive a commission on sales made. Another possible decorator is Overtime that can augment an Employee object so that the employee can receive extra pay if he or she works more than 40 hours/week.

(a) Using the Decorator design pattern, modify the UML model below (by adding, removing, and modifying classes, attributes, operations, associations, multiplicities, etc.) to allow for two optional add-ons that could be applied to any employee: Commission (a percentage of sales made) and Overtime (extra pay if worked more than 40 hours a week). Provide complete operation signatures for the concrete decorator classes.

(b) Explain how the implementation of the Commission::pay(Day, Day) operation is able to retrieve the appropriate range of work records from its Employee object.
Question 6 [15 marks] Design Patterns

Below is a possible partial UML model for email addresses. An email address could represent an individual recipient (shown below). Alternatively, an email address could be an alias for a group, representing a collection of email addresses. An alias for a group consists of a name and a collection of entries, each of which could either be the email address of an individual recipient or could refer to another group of email addresses.

a) Using the **Composite** design pattern, modify the UML model below (by adding, removing, and modifying classes, attributes, operations, associations, multiplicities, etc.) so that the email address ADT that the client uses can represent either the email address of an individual or can represent a group (where a group can include members that themselves are groups, and so on). Include the operations that are expected in a Composite pattern.

b) Using the **Iterator** design pattern, modify the UML model further (by adding, removing, and modifying classes, attributes, operations, associations, multiplicities, etc.) so that clients can create an iterator for iterating through all of the email addresses in a collection of addresses. Include the operations that are expected in an Iterator pattern.

c) Provide implementations of the concrete operations (declared in your model) that the client code uses to create a new iterator.

---

<table>
<thead>
<tr>
<th>EAddr</th>
</tr>
</thead>
<tbody>
<tr>
<td>– userid</td>
</tr>
<tr>
<td>– hostName</td>
</tr>
<tr>
<td>+ EAddr (name:string)</td>
</tr>
<tr>
<td>+ name () : string</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Question</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td><strong>Total</strong></td>
</tr>
</tbody>
</table>