



For the next two questions, consider the following class, which represents a rational number (ratio of two integers like 1/2):

```
class Rational {
    friend ostream& operator>>(ostream& In, Rational& toPrint);
    friend ostream& operator<<(ostream& Out, const Rational& toPrint);
private:
    int Top, Bottom;
public:
    Rational();
    Rational(int T, int B);
    bool operator==(const Rational& RHS) const;
    bool operator<(const Rational& RHS) const;
    bool operator>(const Rational& RHS) const;
    Rational operator+(const Rational& RHS) const;
    Rational operator+(int RHS) const;
    Rational operator-(const Rational& RHS) const;
    Rational operator-(int RHS) const;
    Rational operator*(const Rational& RHS) const;
    Rational operator*(int RHS) const;
    Rational operator/(const Rational& RHS) const;
    Rational operator/(int RHS) const;
};
```

1. [6 points] Assuming all of the member functions of the class `Rational` have been implemented correctly, circle any of the following statements that would not compile.

```
Rational R1(1,1), R2(2,1), R3(3,1), R12(1,2), R13(1,3), R23(2,3);
```

```
Rational SumR, SubR, ProdR, QuotR;
```

```
SumR = R1 + R2 + R3 + R12;
```

```
SubR = 3 * R2 - R23;
```

```
ProdR = R3 * R13 + 4;
```

```
QuotR = R12 / Rational(2, 5);
```

2. [10 points] The mathematical property of commutativity between integers and rationals is not supported on the standard arithmetic operation of multiplication by the C++ `Rational` class. Give the implementation of the multiplication operator that would need to be added to support commutativity.

```
Rational operator*(int LHS, const Rational& RHS) {
    Return (RHS * LHS); // there are harder ways to do this
}
```

For the next three questions, consider the following template classes, which represents a Queue with an underlying double linked list:

```
// DNodeT.h
#ifndef DNODET_H
#define DNODET_H

template <class Item> class DNodeT {
public:
    Item* Data;
    DNodeT<Item> *Next, *Prev;

    DNodeT();
    DNodeT(Item const *newData);
};

// . . . DNodeT template implementation goes here

#endif

// QueueT.h
#ifndef QUEUE_T_H
#define QUEUE_T_H
#include "DNodeT.h"
template <class Foo> class QueueT {
protected:
    DNodeT<Foo> *Front, *Rear; //Head of list == Front of Queue
public:
    QueueT();
    QueueT(const QueueT<Foo>& Source);
    void Enqueue(Foo* const Item);
    Foo* Dequeue();
    bool isEmpty() const;
    QueueT<Foo>& operator=(const QueueT<Foo>& Source);
    ~QueueT();
};
// . . . QueueT template implementation goes here
#endif
```

3. [8 points] Consider a program that declares the following object: `QueueT<string> todoList;`

Is the relationship between `QueueT` and `string` (not `string*`) an association, aggregation or inheritance? Justify your answer.

**The `string` objects stored in the queue are not created by the queue, but by the client code, which then passes the queue pointers to those `string` objects. There is no indication as to whether the `string` objects are deleted by the queue destructor. Even if they are, the independence of their creation suggests an association relationship.**

**(Since the nodes have a `string*` as a data member, that is an aggregation relationship, but that is not the point of this question.)**

4. [6 points] What provision, if any, is made in the queue or node implementations given above to support the derivation of new classes from those?

**Neither template makes use of private data or function members. That will make it easier for derived types to carry out necessary operations on the node and/or queue data.**

**It would be even better (perhaps) if the node and queue provided support for polymorphism by having some virtual functions.**

5. [6 points] What provision, if any, is made in the queue or node implementations given above to support storing a polymorphic data type?

**The data element is secured by a pointer, rather than stored physically. That allows the client to store base-type pointers in the queue/node structure, which is necessary in order to support polymorphic data objects.**

- 
6. [8 points] The designer of a graphical window class for a graphical user interface library is faced with a decision. Certain attributes (e.g., border style) should be the same for all instances of the class, but should also be dynamically changeable. What C++ feature should she use to handle this?

**If all objects of that type share the value of an attribute, it is inefficient for each one to store an individual copy of that attribute. Aside from the storage space, if each object stores an independent copy there is no easy, automatic way to modify all of those at once.**

**Both of these objections can be overcome if the attribute is declared as a static member of the class, and a static mutator is also provided.**

7. [12 points] Consider the two classes below. The designer is faced with one annoying problem. Given the logical significance of the class B, client code should not be able to modify the data member X of B.

```
class A {
protected:
    string X;

public:
    A(const string& iX) {X = iX;}
    string getX() const {return X;}
    void setX(const string& iX) {X = iX;}
};

class B : public A {
public:
    B(const string& iX) : A(iX) {}
};
```

Aside from abandoning the use of inheritance, describe one way the designer could deal with this problem, and describe any shortcomings of your solution.

**The class B could override the inherited function A::setX(). The question is what should the overriding function B::setX() do, and what should its access control be?**

**The implementation of B::setX() must do no harm; the obvious solution is for the implementation to do nothing at all. However, that is somewhat dishonest since the presence of the function would normally imply to a client that it has an obvious effect.**

**If B::setX() is public, then the complaint above applies. However, if B::setX() is declared as either protected or private then client code will not be able to access it at all, and the complaint is removed.**

**It would probably be best to make B::setX() private, since that would also hide its functionality (or lack thereof) from types derived from B as well as from client code.**

8. Consider the template mechanism in C++.

(a) [6 points] Explain why the source code for a template will not compile.

**A template implementation will contain references to its type parameter(s), which are just dummy names. Since those names are used as type names, the compiler will not recognize them as valid and will generate errors.**

(b) [6 points] Explain briefly, in general terms, how the fact stated in (a) is dealt with (since we can, in fact, write useful template implementations and incorporate them in our programs).

**The compiler never sees the template implementation. When a declaration using the template is encountered, the actual type(s) used in that declaration are substituted throughout the template implementation (and other necessary changes are made), resulting in the creation of a new class, which will be compilable.**

9. [12 points] A contemporary C++ programming textbook presents the design of a class called `Insurance`, with the following attributes and behaviors:
- An insurance policy number
  - The name of the person who owns the policy
  - The annual premium
  - Accessors for all the data members
  - Mutators for all the data members
  - Constructors/destructors as needed

The textbook then suggests (as an exercise) "declare two derived classes called `Automobile` and `Home` that inherit the `Insurance` base class". Assume that the proposed name of the class `Automobile` is an accurate name for the intended abstraction. In no more than 100 words, critique the suggestion that `Automobile` be derived from `Insurance`. If you argue that this is not a good idea, explain why not and suggest a reasonable alternative. If you argue that this is a good idea, explain why.

**No, accepting the given class name as accurate, it makes little or no sense to derive `Automobile` from `Insurance`. Certainly an automobile is NOT a kind of insurance, or a specialization of insurance.**

**An `Insurance` object has a number of members, none of which properly belong to an `Automobile`.**

**However, a case can be made that there is, or should be, an association between an automobile and an insurance object.**

**Aggregation is not really an appropriate model, since the insurance isn't part of the automobile, nor is an automobile a part of an insurance object.**

10. Farey numbers are a mathematical construct that somewhat resemble rational numbers. The primary differences are that addition of Farey numbers follows the rule

$$\frac{a}{b} + \frac{c}{d} = \frac{a+c}{b+d}$$

and that the other standard arithmetic operations are not (usually) defined.

- (a) [8 points] Assuming that the `Rational` class presented earlier has been implemented, consider deriving a class `Farey` from `Rational`. Are there any particular reasons that would be difficult or clumsy?

**The `Rational` class has overloaded a rather large number of arithmetic operators, only one of which is appropriate for Farey numbers. While the inappropriate inherited operators could be overridden and rendered harmless, or even inaccessible, that is a clumsy approach to the goal of representing Farey numbers.**

- (b) [6 points] Would those reasons still apply if the hierarchy were reversed. That is, what if a complete `Farey` class were implemented first, and then a `Rational` class were derived from it? Explain.

**Well, this certainly would eliminate the objection to unsuitable inherited members.**

**The only real objection to deriving `Rational` from `Farey` is that so little is gained by doing so. The non-member operators are not inherited anyway. The only member operator of `Farey` would be overridden in `Rational`.**

- 
11. [6 points] What are the three elements that must all be present in order to achieve polymorphism in C++?

**An inheritance hierarchy, virtual functions, and accessing derived type objects via base type pointers.**