Instructions:

- Print your name in the space provided below.
- Answer each question in the space provided.
- If you want partial credit, justify your answers briefly and concisely, even when justification is not explicitly required.
- There are 14 questions, priced as marked. The maximum score is 100.
- When you have completed the test, sign the pledge at the bottom of this page and turn in the test.
- This is a closed-book, closed-notes examination. No calculators or other electronic devices may be used during this examination.
- You may not discuss (in any form: written, verbal or electronic) the content of this examination with any student who has not taken it. You must return this test form when you complete the examination. Failure to adhere to any of these restrictions is an Honor Code violation.

Do not start the test until instructed to do so!

Name

Solution

Pledge: On my honor, I have neither given nor received unauthorized aid on this examination.

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

```cpp
class Rational {
    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.

   ```cpp
   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 = 4 + R1;
   SubR = R2 - R2;
   ProdR = R3 * R13;
   QuotR = R12 / 5;
   ```

   There's no operator+ that takes an int as its left operand and a Rational as its right operand.

2. [10 points] Is the mathematical property of commutativity between integers and rationals supported on the standard arithmetic operation of addition by the above `C++ Rational` class? If not, give the prototype and implementation of the addition operator that would need to be added to support commutativity.

   ```cpp
   friend Rational operator+(int LHS, const Rational& RHS);

   Rational operator+(int LHS, const Rational& RHS) {
       return (Rational(LHS * RHS.Bottom + RHS.Top, RHS.Bottom));
   }
   ```

   That's the most obvious solution, however there are others.

For the next three questions, consider the following template classes, which represents a Queue with an underlying double linked list:
3. [5 points] Is the relationship between QueueT and LinkNodeT an association, aggregation or inheritance? Justify your answer.

Aggregation. LinkNodeT objects are created only by QueueT member functions, and are deleted by the QueueT destructor (or should be). Put another way, you can't have a queue without having nodes (although you could perhaps have a node without having a queue).
4. [10 points] Assume that all of the functions of both classes have been implemented, except for the copy constructor and assignment operator overload. Give the C++ code to implement the assignment operator overload according to the above class declarations:

```cpp
QueueT<Foo>& operator=(const QueueT<Foo>& Source) {
    if ( this == &Source ) return (this); // self-assignment?
    while ( !isEmpty() ) { // clean up LHS
        Dequeue();
    }
    Front = Rear = NULL;
    LinkNodeT<Foo>* Curr = Source.Front; // hook into Source list
    while ( Curr != NULL ) { // copy Source list data
        Enqueue(Curr->getData()); // retrieve data and insert
        Curr = Curr->getNext(); // advance hook
    }
    return (*this); // return LHS by reference for chaining
}
```

Notes:

1. Because the parameter is const, you can't call Enqueue() or Dequeue() on Source.

2. You can't use the copy constructor (since it hasn't been implemented) or the assignment operator (since this IS the implementation of =) to make a local copy of Source and work on it to avoid const.

3. The assignment operator is acting on the object on the LHS of the assignment, not on a local object. If you declare a local QueueT<Foo> and operate on it, you aren't modifying the right object.

4. You CAN implement this by using the low-level LinkNodeT functions, but it's much more complicated.

5. [6 points] For the above QueueT template class, briefly explain the implementation differences between the assignment operator overload and the copy constructor?

The copy constructor does not need to:
- clean up a target object since it's initializing newly-allocated memory
- check for identity between the source object and a target
- return a value

(It's also very possible the assignment operator didn't really need to set Front and Rear to NULL since Dequeue() really should do that when the queue becomes empty.)
For the next two questions, consider the following classes:

```cpp
#include <string>
using namespace std;

class Point {
private:
    double X, Y;
public:
    Point(double iX = 0.0, double iY = 0.0) {X = iX; Y = iY;}
    double getX() const {return X;}
    double getY() const {return Y;}
    void setPoint(double iX = 0, double iY = 0) {X = iX; Y = iY;}
    ~Point() {};
};
class Circle {
private:
    Point Center;
    double Radius;
public:
    Circle(Point C = Point(4.2, 3.1), double iR = 0.5) {Center = C; Radius = iR;}
    Circle& setCenter(Point C) {Center = C; return *this;}
    Circle& setRadius(double iR) {Radius = iR; return *this;}
    Point getCenter() const {return Center;}
    double getRadius() const {return Radius;}
    double Area() const {return (PI * Radius * Radius);};
    ~Circle() {};
};
const double PI = 3.14159;
```

6. [4 points] If one makes the following declaration:

```cpp
Circle X;
```

What is the center of X or is it random?

The Circle constructor takes default values for both of its parameters; therefore, X.Center will be (4.2, 3.1).

7. [9 points] Given the following code segment to execute:

```cpp
Point P(7, 11);
Circle Hoop(P, 3.14);
Point Q(0, 3);
Hoop.setCenter(Q).setRadius(2.72);
cout << Hoop.getCenter().getX() << " " << Hoop.getCenter().getY() << endl;
cout << Hoop.getRadius() << endl;
```

What is output by the above code?

Hoop.setCenter(Q) will set Hoop.Center to the point (0, 3). Since Circle::setCenter() returns its host object by reference, the call to setRadius() will also act on Hoop, setting Hoop.Radius to 2.72. So the output code will print the values 0 and 3 and 2.72.
For the next 5 questions, consider the following class, which builds upon the classes from the previous two questions:

```cpp
#include <cmath>

class Cone : public Circle {
private:
    double Height;
public:
    Cone(Circle Base, double iH) : Circle(Base) {Height = iH;}
    Cone& setHeight(double iH) {Height = iH;}
    double getHeight() const {return Height;}
    double SlantHeight() const {
        return (sqrt(getRadius() * getRadius() + Height * Height));
    }
    double Area() const {
        return (PI * getRadius() * getRadius() +
                PI * getRadius() * SlantHeight());
    }
    double Volume() const {
        return (PI * getRadius() * getRadius() * Height / 3.0);
    }
    ~Cone() {}};
```

8. [8 points] Can one make the following declaration?

```cpp
Cone A[10];
```

Briefly explain?

No. To declare an array of Cone objects, the class Cone must have a default constructor. It does not. That's because Cone has a constructor that takes two parameters, which prevents the automatic generation of a default parameterless constructor.

9. [8 points] Given the following declaration:

```cpp
Cone Z(Circle(Point(3,3), 3.14159), 2.71828);
```

Can one make the following member invocation, (briefly explain)?

```cpp
Point pc = Z.getCenter();
```

Yes. Circle::getCenter() is public, and so can be called by any member of a class derived from Circle using inheritance.

For the next 2 questions, consider the following alternative implementation of Cone::Volume():

```cpp
double Volume() const {
    return (PI * Radius * Radius * Height / 3.0);
}
```

10. [8 points] Explain why this alternative implementation is not valid, (the above formula is OK).

Circle::Radius is private in Circle and so is inaccessible to members of a class derived from Circle.
11. [8 points] Aside from eliminating inheritance, what change would make this alternative implementation valid, (without changing the alternative implementation code itself)?

Circle could declare Radius as protected (or public, although that's an abomination).

12. [8 points] Briefly explain how Base is being used in the constructor for Cone?

Base is used to initialize the Circle "layer" of the Cone object, using the (automatic) copy constructor for the Circle class.

For the next two questions, consider the following function:

```cpp
void check() throw(exception) {
    try {
        throw exception(); //throw a standard exception object
    } catch (exception& e) {
        cout << e.what() << endl;
        check();
        throw(e);
    }
}
```

13. [6 points] Assume that check has been called from main(). Given that the standard exception class provides a member function what() that returns a string describing the exception error, which of the following statements about the above function is true?

1) The code will not compile since it is illegal to throw an exception within a catch clause.
2) The code will not compile since it is illegal to throw an exception before the function body.
3) If the function that invokes check() (i.e. main()) does not catch the exception the program will abort.
4) The invocation of check() will be removed from the runtime stack as soon as the exception is thrown.
5) The invocation of check() will result in infinite recursive calls to check() being made.
6) None of the above, they are all false statements.

It is perfectly legal to catch and rethrow an exception. The "throw" before the function body is not an executable statement; it's a qualifier specifying what the functions is allowed to throw. The caller will never have a chance to catch the exception. Since check() catches the original exception itself, it is not popped from the runtime stack.

14. [4 points] How many times will the throw(e); statement in the above function be executed?

1) 1
2) 2
3) 3
4) infinite, \(\infty\)
5) 0
6) \(\leq \text{INT\_MAX}\)

Given the answer to question 13, we know that this statement will never be reached.