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 8 questions, priced as marked. The maximum score is 100.
- Legibility counts. Responses that cannot be easily read may not be graded!
- 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 ___________________________ printed

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

______________________________ signed
1. [15 points] No single built-in type C++ adequately models the Day, Month, Year representation of a date. Develop an interface, (not implementation and not representation), to provide the natural, common operations for a Date class. From a client's perspective, a date object will simply represent a Day, Month and Year for a specific date. The interface should not be influenced by any underlying representation design decisions. Carefully consider justifications for your chosen interface. It should be complete, but do not include support for behaviors that do not make sense. Nonsensical interface elements will be penalized. You may assume that any date to be stored is a Gregorian calendar date, (i.e., no conversion between different types of calendar dates is required). The Date class interface must be given as a list of C++ public method prototypes.

```cpp
class Date {
private:
    //to be designed at a later date

public:
    //constructors
    Date(unsigned int Mon, unsigned int Day, unsigned int Year);
    Date(string date);  //optional
    //accessors
    unsigned int getMonth() const;
    unsigned int getDay() const;
    unsigned int getYear() const;
    string getDate() const; //optional
    //mutators - optional
    bool setDate(unsigned int Mon, unsigned int Day, unsigned int Year);
    bool setMonth(unsigned int Mon);
    bool setDay(unsigned int Day);
    bool setYear(unsigned int Year);
    //transformers - optional
    string dayOfWeek() const;
    bool leapYear() const;
    Date operator++();    //or Date& next();
    Date operator++(int); //optional
    Date operator--();    //or Date& prev();
    Date operator--(int); //optional
    Date operator+(unsigned int Days);
    Date operator-(unsigned int Days);
    int operator-(const Date& D) const;
    bool operator==(const Date& D) const;
    bool operator!=(const Date& D) const;
    bool operator<(const Date& D) const;
    bool operator<=(const Date& D) const;
    bool operator>(const Date& D) const;
    bool operator>=(const Date& D) const;
    ~Date(); //optional

friend ostream& operator<<(ostream& out, const Date& D);
friend istream& operator>>(istream& in, Date& D);
} //end Date
//commutative addition
Date operator+(const Date& D, unsigned int Days);  //optional
```
For the next 3 questions, consider the following painfully incomplete class declaration and painful implementation:

```cpp
class Odometer {
private:
    double miles;
    double round2tenths(double m);
public:
    Odometer(double m = 0.0);
    double mileage();
    Odometer& travel(double m = 0.0);
    Odometer rollBack(double m = 0.0);
};
Odometer::Odometer(double m) {
    miles = round2tenths(m);
}

double Odometer::mileage() {
    return( miles );
}

Odometer& Odometer::travel(double m) {
    miles += round2tenths(m);
    return( *this );
}

Odometer Odometer::rollBack(double m) {
    if (round2tenths(m) < miles)
        miles -= round2tenths(m);
    else
        miles = 0.0;
    return( *this );
}

double Odometer::round2tenths(double m) {
    return( floor(fabs(m) * 10.0 + 0.5) / 10.0 );
}
```

2. [10 points] What would be output by the following code?

```cpp
Odometer lemon(250000.0), lateModel(10000.0);
lemon.travel(33000).rollBack(200000.0);
lateModel.rollBack(5000).travel(4000);
cout << lemon.mileage() << " miles: Lemon " << endl
    << lateModel.mileage() << " miles: Late Model " << endl;
```

```
OUTPUT
83000.0 miles: Lemon
5000.0 miles: Late Model
```

The rollBack() method returns a modified copy of lateModel on which the travel() function is invoked which changes the copy and NOT the

```
83000.0 miles: Lemon
5000.0 miles: Late Model
```
3. [10 points] In order to avoid the overhead of the accessor calls in the code in the previous question, give the prototypes, (and only the prototypes – not the implementations), necessary to allow code such as the following:

```cpp
Odometer turkey(250000.0), showroom;

    cout << turkey << " miles: Turkey " << endl;
    cin >> showroom;
```

```cpp
friend ostream& operator<<(ostream& out, const Odometer& od);
friend istream& operator>>(istream& in, Odometer & od);
```

4. [15 points] Note that the above Odometer class is missing, (a lot of things), an overloading of the C++ division operator to allow Odometer objects to be divided by nonnegative integral values in order to compute yearly, monthly, weekly or daily mileage averages. Give the function prototype and implementation, to overload the C++ division operator for Odometer objects and nonnegative integral values. Be sure to allow appropriate chaining of expressions involving Odometer objects and allow for commutative division operations.

```cpp
//prototype
do_text
  operator/(unsigned int time) const;
// OR
Odometer operator/(unsigned int time) const;
```

```cpp
//implementation
Odometer::operator/(unsigned int time) const {
    if ( time > 0 )
        return (round2tenth( miles / double(time) ) );
    else
        return ( UINT_MAX );
}
```
Consider the product description below:

The Braun Syncro Razor System provides the user with LCD display of charge time remaining, shave time, cleaning time, clean and oil indicator. This electric razor's unique 4 way moving head captures even more hairs in fewer strokes for ultimate closeness. The automatic self-cleaning Clean&Charge system leaves your shaver thoroughly and hygienically clean for a better shaving performance every day. The alcohol based cleaning fluid is pumped through the shaver head so it quickly and easily cleans away all the shaved stubbles and skin grease without any mess in the sink. The shaved stubbles are collected in the refill. The filter inside the Clean&Charge keeps the shaved stubbles in the refill. This Syncro razor is enhanced with the smart chip technology, which is activated every time the shaver is charged, and maintains the battery power at its peak.

Features include:
- Interactive LCD panel: The interactive LCD display provides useful data on charge status and maintenance in a form that is easy to understand, detailed and practical. The display shows the operation mode and charging conditions of the shaver and if the shaver has to be cleaned or lubricated.
- Includes charging and low charge indicator
- 4 Way moving head: Oscillates to capture more hair in fewer strokes, as well as pivots to follow contours of the neck and face.
- Syncro Clean and Charge System (look above for details)
- Ultra speed motor: Includes Braun's best motor with 8,500 rpm.
- Cord/Cordless operation: 1 hour recharge for up to 50 minutes of cordless shaving with a 3 minute quick charge
- Sliding long hair trimmer
- Self cleaning and charging system allows for hands free cleaning without dismantling of shaver.
- Shaver remains safely stored in unit after clean/charge cycle
- Cleaning solution: replaceable cartridge - easy to clean disposal of shaving debris; alert flashes when refill needs replacement

5. [15 points] Identify a reasonable set of classes, (but not relationships), for the modeling of this product. Give a descriptive name and a one-line description of the purpose of each class. Your analysis leading to the set of classes will not be graded, only the end result. You should apply some structured process, such as that of Abbott and Booch.

| Classes Optional classes |
|--------------------------|--------------------------------------------------|
| Razor System: (system aggregation class) | Charging indicator, low charge indicator, refill alert |
| LCD display: | |
| Head | |
| Motor | |
| Cord | |
| Longhair Trimmer | |
| Unit – storage | |
| Solution cartridge | |
| Pump | |
| Clean&Charge system: | Refill, Filter |
| Smart Chip | |
| Battery | |
| | Fluid |
Consider the following interface for the `DMSAngle` class:

```cpp
enum Sign {NONNEGATIVE, NEGATIVE};  //Line 1

class DMSAngle {
    friend std::ostream& operator<<(std::ostream& Out, const DMSAngle& Angle);  // 2
    friend std::istream& operator>>(std::istream& In, DMSAngle& Angle);  // 3

public:   // 4
    DMSAngle(Sign Sn = NONNEGATIVE, unsigned long Deg = 0, unsigned char Min = 0, unsigned char Sec = 0);  // 5
    DMSAngle(double Sec);  // 6
    DMSAngle operator+(const DMSAngle& RHS) const;   // 8
    DMSAngle operator-(const DMSAngle& RHS) const;   // 9
    DMSAngle operator+() const;   //10
    DMSAngle operator-() const;   //11
    DMSAngle operator*(double Mult) const;        // mult. by a scalar  //12
    DMSAngle operator/(double Div) const;         // division by a scalar  //13
    double operator/(const DMSAngle& RHS) const;  // ratio of angles   //14
    DMSAngle operator-() const;                   // complement of angle  //15
    bool operator^ (const DMSAngle& RHS) const;    // ? coterminous angles  //16
    bool operator==(const DMSAngle& RHS) const;   //17
    bool operator!=(const DMSAngle& RHS) const;   //18
    bool operator<(const DMSAngle& RHS) const;   //19
    bool operator<=(const DMSAngle& RHS) const;   //20
    bool operator>(const DMSAngle& RHS) const;   //21
    bool operator>=(const DMSAngle& RHS) const;   //22

private:   //23
    double Seconds;  //24
};  //25
```

Given the following valid declarations:

```cpp
DMSAngle A0, A1(NONNEGATIVE, 45, 0, 0), A2(NEGATIVE, 90, 0, 0), A3(NONNEGATIVE, 180, 30, 30), A4(-360), A5(108000);
```

6. [10 points] Assuming all of the member functions of the above class `DMSAngle` have been implemented correctly, circle any operator in the following statements that would break compilation.

```cpp
A0 = A1 / -A2 / A3;       // A1 / -A2 results in a double / A3 which fails
A0 = 3.0 * A2 * 4.0;     // 3.0*A2 results in a DMSAngle*4.0 which is ok
A0 = -A3 / A4 * A2;      // ~A3/A4 results in a double*A2 which is ok
```

cout << boolalpha << (+~A4 ^ A3);  //+-~A3 results in a DMSAngle^A3

DMSAngle A6 = -60.0;       // equivalent to DMSAngle A6(-60.0); which is ok
For the next two questions, consider another interface variation for the Odometer class:

```cpp
class Odometer {
private:
  unsigned char  *miles; //array pointer
  unsigned int   numDigits; //array dimension
  double round2tenth(double m);
public:
  Odometer(double m = 0.0);
  double mileage();

  //_______________________________; //copy constructor
  //_______________________________; //assignment operator
  Odometer& travel(double m = 0.0);
  Odometer rollBack(double m = 0.0);
};
```

Note that the underlying implementation representation has been changed to a dynamic unsigned char array whose dimension equals the number of digits in the mileage values stored. Each array location will store one digit of the odometer, with the zero position storing the tenth’s of a mile digit, the one position storing the one’s mile digit, etc.

7. [10 points] If you believe that they are necessary for logically correct operation of the class, give the prototypes for the copy constructor and assignment operator. If you do not believe they are necessary briefly, explain why.

Odometer (const Odometer& od);  //copy constructor
Odometer& operator=(const Odometer& od);  // assignment operator

8. [15 points] Give the implementation for the copy constructor for the above Odometer class variation, (implement it whether you believe it is necessary or not for the logically correct operation of the class).

```cpp
Odometer (const Odometer& od) {  //copy constructor
  numDigits = od.numDigits;    //copy non-dynamic members
  if ( (numDigits == 0) || (od.miles == NULL) )  //no dynamic memory
    miles = NULL;
  else {
    miles = new(nothrow) unsigned char [numDigits]; //allocate memory
    for (int i=0; i < numDigits; i++) //deep copy
      miles[i] = od.miles[i];
  }
}
```