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 a Va Tech Honor Code violation, (www.honorsystem.vt.edu).

Do not start the test until instructed to do so!

Circle your instructor and section:

Barnette     Wheaton
8:00AM TuTh  11:00AM TuTh  10:10AM MWF  12:20PM MWF

Name ___________________________ printed

PID ___________________________ VT email address (NOT SSN) - printed

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

______________________________ Signed
For the next 2 questions, consider the following simple class declaration and implementation:

```cpp
class BasketBallGameScore {
private:
    unsigned int score;
public:
    BasketBallGameScore(unsigned int S=0);
    unsigned int Score() const;
    BasketBallGameScore FoulShot();
    BasketBallGameScore Pointer2();
    BasketBallGameScore Pointer3();
};

BasketBallGameScore::BasketBallGameScore(unsigned int S) {
    score = S;
}

unsigned int BasketBallGameScore::Score() const {
    return score;
}

BasketBallGameScore BasketBallGameScore::FoulShot() {
    ++score;
    return *this;
}

BasketBallGameScore BasketBallGameScore::Pointer2() {
    score += 2;
    return *this;
}

BasketBallGameScore BasketBallGameScore::Pointer3() {
    score += 3;
    return *this;
}
```

1. [10 points] Consider that the above class will be used to implement a scoreboard for a basketball game. Given the possible situation where the official scorer notifies the scoreboard operator that a mistake has been made in the score. Using the above `BasketBallGameScore` class interface is it possible to set a `BasketBallGameScore` object to a corrected score? If your response is yes, then give a short segment of client code to do so, If your response is no, then briefly explain your answer.

   Yes, using the default assignment operator

   ```cpp
   BasketBallGameScore VT;
   //... operations on VT
   int correction = -1;
   VT = BasketBallGameScore ( VT.Score() + correction );
   ```
2. [10 points] What would be output by the following code?

```cpp
BasketBallGameScore VT, Prov;
VT.Pointer3();
VT.Pointer2(); // VT now has 5 points
VT.FoulShot();
cout << VT.Score() << " Score after single Foul Shot" << endl;
VT.FoulShot().FoulShot();
cout << VT.Score() << " Score after 2 more Foul Shots" << endl;
```

**OUTPUT**

```
6 Score after single Foul Shot
7 Score after 2 more Foul Shots
```

The FoulShot() method returns a modified *copy* of BasketBallGameScore on which the second FoulShot() function is invoked.

3. [15 points] Note that the above BasketBallGameScore class is missing overloads of C++ operators to allow BasketBallGameScore objects to be used in expressions such as the following:

```cpp
BasketBallGameScore home, visitor;
unsigned int homeAverageScoreByHalf = home / 2;
unsigned int AverageGamePointsScoredPerQuarter = (home + visitor) / 4;
```

Give the function prototype(s) and implementation(s), to overload the necessary C++ operators for BasketBallGameScore objects and any necessary primitive types to allow expressions of the form shown above.

```cpp
unsigned int operator/(unsigned int div) const;
BasketBallGameScore operator+(const BasketBallGameScore& RHS) const;
____________________________________________
unsigned int BasketBallGameScore::operator/(unsigned int div) const {
    if ( div == 0 )
        return( UINT_MAX );
    return( score / div );
}

BasketBallGameScore BasketBallGameScore::operator+(const BasketBallGameScore& RHS) const {
    return(BasketBallGameScore( score + RHS.score );
}
Consider the following interface for the YFIDistance class:

```cpp
class YFIDistance {
    friend std::ostream& operator<<(std::ostream& Out, const YFIDistance& Dist);
    friend std::istream& operator>>(std::istream& In, YFIDistance& Dist);

public:
    YFIDistance(unsigned long Yd = 0, unsigned long Ft = 0,
                unsigned long In = 0);
    YFIDistance(double In);
    YFIDistance operator+(const YFIDistance& RHS) const;
    YFIDistance operator-(const YFIDistance& RHS) const;
    YFIDistance operator+() const;
    YFIDistance operator*(double Mult) const;  // mult. by a scalar
    YFIDistance operator/(double Div) const;  // division by a scalar
    YFIDistance operator*(double Mult, const YFIDistance& RHS);  
    YFIDistance operator/((double Mult, const YFIDistance& RHS);  

private:
    double Inches;
};
```

Given the following valid declarations:

```
YFIDistance A0, A1(45, 2, 11), A2(90, 1, 6), A3(180, 0, 3);
```

4. [10 points] Assuming all of the member functions of the above class YFIDistance have been implemented correctly, circle any operator, (not just the expression), in the following statements that would break compilation.

```
A0 = A1 + +A2 ; // A1 + +A2 results in a
    // YFIDistance + YFIDistance which is ok
A0 = 2.0 * A1 * 3.0; // 2.0 * A1 * 3.0 results in double * YFIDistance
    // yielding a YFIDistance * a double also yielding
    // a YFIDistance which is ok
A0 = A1 / 2.0 / 3.0; // A1 / 2.0 / 3.0 results in YFIDistance / double
    // yielding a YFIDistance / a double also yielding
    // a YFIDistance which is ok

cout << boolalpha << ( (YFIDistance(A1 / A2)) != (A3) );
// (A1 / A2) results in a double constructed into a YFIDistance compared //
// for inequality to A3 which is ok
A0 = A1 * A2 / A3; // A0 = A1 * A2 / A3 results in YFIDistance *
    // YFIDistance yielding a double / a YFIDistance
    // which fails
```
For the next two questions, consider the following declaration for a `YFIDistanceStack` class:

```cpp
#include "YFIDistance.h"

typedef YFIDistance* YFIDistancePtr;

class YFIDistanceStack {
private:
    YFIDistancePtr* DistanceStk; //array pointer <=> YFIDistance** DistanceStk
    unsigned int Capacity;      //array dimension
    unsigned int Size;          //array size
public:
    YFIDistanceStack(unsigned int Cap=0);
    //_______________________________; //copy constructor
    //_______________________________; //assignment operator

    bool Push(YFIDistance d);
    bool Pop();
    YFIDistance Top() const;
    bool Empty() const;
    ~YFIDistanceStack();     //destructor
};
```

Note that the underlying implementation representation is a dynamic array of pointers to `YFIDistance` objects.

5. [10 points] Give the prototypes for the copy constructor and assignment operator.

```cpp
YFIDistanceStack(const YFIDistanceStack& Source); //copy constructor

YFIDistanceStack& operator=(const YFIDistanceStack& Source); // = overload
```
6. [15 points] Give the implementation for the copy constructor for the above `YFIDistanceStack` class.

```cpp
YFIDistanceStack(const YFIDistanceStack& Source) {

    Capacity = Source.Capacity;
    Size = Source.Size;

    if ( (! Source.DistanceStk) || (Capacity == 0) ) { //null array check
        DistanceStk = NULL;
        return;
    } //endif

    //allocate array of pointers
    DistanceStk = new (nothrow) YFIDistancePtr[Capacity];

    if (! DistanceStk) { //heap exhausted
        Capacity = Size = 0;
        return;
    } //endif

    for(unsigned int i=0; i++; i < Size) {
        DistanceStk[i] = new(nothrow) YFIDistance( *(Source.DistanceStk[i]) );
        //if (DistanceStk[i] == NULL) //heap exhausted
        //break;
    }

    for(unsigned int i=Size; i < Capacity; i++) //zero out remaining array cells
        DistanceStk[i] = NULL;
}
```
//Alternate solution

YFIDistanceStack(const YFIDistanceStack& Source) {

    Capacity = Source.Capacity;
    Size = Source.Size;

    if ( (! Source.DistanceStk) || (Capacity == 0) ) { //null array check
        DistanceStk = NULL;
        return;
    } //endif

    //allocate array of pointers
    DistanceStk = new (nothrow) YFIDistancePtr[Capacity];

    //allocate all object copies at once
    YFIDistance* YFIobjs = new(nothrow) YFIDistance[Size];

    if ( (! DistanceStk) || (! YFIobjs)) { //heap exhausted
        if (DistanceStk) delete [] DistanceStk;
        if (YFIobjs) delete [] YFIobjs;
        DistanceStk = NULL;
        Capacity = Size = 0;
        return;
    } //endif

    for(unsigned int i=0; i++; i < Size) {
        FYIObj[i] = *(Source.DistanceStk[i]); //copy source object
        DistanceStk[i] = &(FYIObj[i]);
    }

    for(unsigned int i=Size; i < Capacity; i++) //zero out remaining array cells
        DistanceStk[i] = NULL;
}
7. [15 points] No single built-in C++ type that adequately models a Bank Account for representing checking, savings and other accounts. Develop an interface, (not implementation and not representation), to provide natural, common operations for a BankAccount class. From a client’s perspective, a BankAccount object will simply represent a person’s account balance. The interface should not be greatly 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 only operations upon U.S. currency are to be supported, (i.e., no conversion between different types of currency is required). The BankAccount class interface must be given as a list of C++ public method prototypes. [Do NOT give implementation code.]

```cpp
class BankAccount {
private:
    //to be designed later
    BankAccount& operator=(const BankAccount& RHS); //prevent assignment
public:
    BankAccount(unsigned int AccountID, const String& Last, const String& First, const String& Middle="", AccountType at = CHECKING, double Balance = 0.0);
    BankAccount(unsigned int AccountNum, const String& Last, const String& First, const String& Middle="", AccountType at = CHECKING, unsigned int TotalCents = 0.0);
    BankAccount(unsigned int AccountNum, const String& Last, const String& First, const String& Middle="", AccountType at = CHECKING, unsigned int Dollars = 0.0, unsigned int Cents = 0.0);
    ~BankAccount();

    bool operator==(const BankAccount& RHS) const;
    bool operator!=(const BankAccount& RHS) const;
    bool operator<(const BankAccount& RHS) const;
    bool operator>(const BankAccount& RHS) const;
    bool operator<=(const BankAccount& RHS) const;
    bool operator>=(const BankAccount& RHS) const;
    double Balance() const;
    unsigned int AccountID() const;
    string Name() const;
    AccountType typeAccount() const;

double operator+(const BankAccount& RHS) const; //combined shares
    double operator+(double deposit);
    double operator+(unsigned int depositCents); //optional
    double operator-(double withdrawal);
    double operator-(unsigned int withdrawalCents); //optional
    double operator*(double interestRate) const;
    double operator/(unsigned int days); //optional – avg daily balance

friend istream& operator>>(istream& in, BankAccount& BA);
friend ostream& operator<<(ostream& out, const BankAccount& BA);

friend double operator+(double deposit, BankAccount& BA);
friend double operator-(double withdrawal, BankAccount& BA);
friend double operator*(double interestRate, const BankAccount& BA);
}
```
8. [15 points] Given the description below for an electric toothbrush, 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.

The Sonicare sonic toothbrush cleans with a combination of 31,000 brush strokes per minute and gentle sonic waves. Reduces plaque bacteria in hard-to-reach areas between teeth. Advance model features "Smartimer" Automatic 2-minute shut-off helps ensure dentist-recommended brushing time. It also features a quadtimer which "beeps" every 30 seconds to allow the user to move quadrants and evenly distribute the two-minutes brushing time. Distinctive "Easy-start" automatic ramp-up feature gently increases brushing power during the first twelve uses, helping you get accustomed to electric toothbrushing. Comes with an integrated charging base with tower for ventilated storage of 2 brushheads. A charge indicator light displays red while the battery is under charge and green when fully charged. "Better Checkup" guarantee - Manufacturer guarantees a better dental checkup after using any Sonicare toothbrush for 90 days or you'll receive a full refund.

We are not interested in the interfaces of any possible objects contained within the toothbrush. But, if contained objects do exist they may impact the interface you design for the toothbrush.

<table>
<thead>
<tr>
<th>Classes</th>
<th>Optional classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sonic Toothbrush: (system aggregation class)</td>
<td></td>
</tr>
<tr>
<td>Brush (motor)</td>
<td>Beeper/Sound</td>
</tr>
<tr>
<td>Charger / Base</td>
<td></td>
</tr>
<tr>
<td>Battery</td>
<td></td>
</tr>
<tr>
<td>Sonic Waves (generator/controller)</td>
<td></td>
</tr>
<tr>
<td>Smart Timer (could be listed with Quad Timer as a sub-Timer class)</td>
<td></td>
</tr>
<tr>
<td>Quad Timer</td>
<td></td>
</tr>
<tr>
<td>Rampup / Easy-start</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Tower</td>
</tr>
</tbody>
</table>