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 12 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!

---

KEY

Name ____________________________ printed

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

______________________________ signed
1. [5 points] State in one brief sentence the primary difference between the Procedural paradigm and the Object-Oriented paradigm.

   Procedural designed systems are task centered, Object-Oriented designed systems are data centered.

2. [5 points] Considering the Taxonomy of Member Functions covered in lecture, (Constructors, Mutators, Observers/Reporters, and Iterators), const member functions in a class would always be classified in what taxonomy category?

   Observers/Reporters (because const member functions cannot change data members).

3. [5 points] The software structures objects and classes support what design strategies and which software engineering goals?

   Design Strategies: abstraction, separation and composition.
   Software Eng. Goals: reusability

4. [15 points] Given the description below for a Kerosene Heater, design the public interface for a class that models this heater. Your design should not list any data members or private functions. Specifically your design should only include a list of services the class provides for the client. For each such service specify the parameters and the response, if any.

   The Model 105 kerosene heater from WeeHeatEm represents the state of the art in portable heating technology. The unit features an easy-on electronic ignition for quick, reliable starting, and may be turned off manually with a simple twist of one knob. You may select any of three available heating levels, from 5,000 to 10,000 to 24,000 BTUs of heat. The Model 105 will burn for up to 18 hours on a single tank (1.99 gallons) of kerosene. A gauge with a magnified dial provides an accurate indication of the fuel level. Safety features include a suspended grill housing to prevent casual burns. The top surface of the grill may be used to hold a teakettle or pot for cooking. A tip-over detector automatically shuts off the unit if it is tilted more than 20 degrees from vertical.

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

<table>
<thead>
<tr>
<th>Response</th>
<th>Service</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Success/failure</td>
<td>bool</td>
<td>on()</td>
</tr>
<tr>
<td>success/failure</td>
<td>bool</td>
<td>off()</td>
</tr>
<tr>
<td>BTULevel</td>
<td>setHeatLevel</td>
<td>(BTULevel btus)</td>
</tr>
<tr>
<td>FuelQuantity</td>
<td>double</td>
<td>int</td>
</tr>
<tr>
<td>FuelAdded</td>
<td>FuelQuantity</td>
<td>double</td>
</tr>
</tbody>
</table>
For the next 3 questions, consider the following woefully incomplete class declaration and implementation:

```cpp
class Weight {
private:
    unsigned int pds, oz;
public:
    Weight(unsigned int p=1, unsigned int z=0);
    Weight gain(unsigned int p, unsigned int z);
    Weight loss(unsigned int p, unsigned int z);
    unsigned int getPds() const;
    unsigned int getOz() const;
};
Weight::Weight(unsigned int p, unsigned int z) {
    pds = p; oz = z;
}
Weight Weight::gain(unsigned int p, unsigned int z) {
    return (Weight(pds + p, oz + z));
}
Weight Weight::loss(unsigned int p, unsigned int z) {
    return (Weight(pds - p, oz - z));
}
unsigned int Weight::getPds() const {
    return (pds);
}
unsigned int Weight::getOz() const {
    return (oz);
}
```

The gain() method returns a modified copy of littleJohn on which the getPds function is invoked. However the getOz() function is invoked upon the littleJohn object which has not been changed.

5. [5 points] What would be output by the following code?

```cpp
Weight littleJohn(300, 0);
cout << littleJohn.gain(50, 8).getPds() << " lb "
    << littleJohn.getOz() << " oz" << endl;
```

**OUTPUT**

350 lb 0 oz

6. [10 points] Note that the above Weight class is missing an overloading of the C++ multiplication operator to allow Weight objects to be multiplied by nonnegative integers. Give the function prototype, (and only the prototype – not the implementation), to overload the C++ multiplication operator, for Weight objects and integers.

```cpp
Weight operator*(unsigned int RHS) const;
```

With an eye towards promoting information hiding, give a second prototype to make the multiplication operator commutative.

```cpp
Weight operator*(unsigned int LHS, const Weight& w);
```
7. [10 points] In order to avoid the overhead of the accessor calls in the code in the previous question, give the prototype and implementation of additional functionality necessary to allow code such as the following:

```cpp
Weight littleJohn(300, 0);
cout << littleJohn << endl;
friend ostream& operator<<(ostream& out, const Weight& w); //prototype
ostream& operator<<(ostream& out, const Weight& w) {
    out << w.pds << " lb " << w.oz << " oz";
    return out;
}
```

Consider the product description below:

**Buzz Lightyear Talking Room Guard**

- As a leader of the elite Universe Protection Unit, Buzz Lightyear protects your sector of the galaxy from the threat of invasion
- His mission: guarding your room with radar motion sensor and blasting off intruders with his laser
- When the intergalactic security system is activated, clearance is granted only to those with the correct secret pass code (enclosed)
- Over 35 phrases and sound effects

If intruders don't want to get on Buzz Lightyear's bad side, then they'd better stay outside. And who better to protect your galaxy sector (or bedroom) than the leader of the Universe Protection Unit himself, featured in the hit movie Toy Story. Standing almost 14 inches tall on his black-and-silver pedestal, our rugged plastic hero is a battery-powered (two AA batteries included) protector, as well as a swell conversationalist. His trio of voice sampler buttons elicits anything from "I'm Buzz Lightyear; I come in peace" to activation of the motion control sensor. Get too close while Buzz is in activation mode ("The Intergalactic Security System will be armed in 5 seconds; please leave this sector!"), and he'll not be shy about telling you that it's "Time to blast off!" He'll stay on the job, too, until deactivated using your secret button code sequence. Not that he's all talk: additional buttons activate laser sounds and spring-release jetpack wings, too. Articulated arm joints and tilting face shield suggest that if there's anything Buzz can't do, one might have to search infinity and beyond to find it.

8. [10 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. Nevertheless, you should apply some structured process, such as that of Abbott and Booch.

<table>
<thead>
<tr>
<th>Classes</th>
<th>Optional classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>BuzzLYtalkingRoomGuard</td>
<td>Arm Joints</td>
</tr>
<tr>
<td>RadarMotionSensor</td>
<td>Face Shield</td>
</tr>
<tr>
<td>SecuritySystem</td>
<td>JetPackWings</td>
</tr>
<tr>
<td>Sound</td>
<td></td>
</tr>
<tr>
<td>Buttons</td>
<td></td>
</tr>
<tr>
<td>Battery</td>
<td></td>
</tr>
<tr>
<td>Laser</td>
<td></td>
</tr>
</tbody>
</table>
Consider the following interface for the `Money` class:

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

class Money { // 2
public: // 3
    Money(); // 4
    Money(int TotalCents); // 5
    Money(unsigned int TotalCents); // 6
    Money(double Amount); // 7
    Money(int Dollars=0, int Cents=0); // 8
    ~Money(); // 9

    bool operator==(const Money& RHS) const; //10
    bool operator!=(const Money& RHS) const; //11
    bool operator<(const Money& RHS) const; //12
    bool operator<=(const Money& RHS) const; //13
    bool operator>(const Money& RHS) const; //14
    bool operator>=(const Money& RHS) const; //15

    Money operator+(const Money&) const; //16
    Money operator-(const Money&) const; //17
    Money operator*(int Factor) const; //18
    Money operator*(double Factor) const; //19
    Money operator/(int Divisor) const; //20
    Money operator-() const; //21
    Money operator+() const; //22

friend istream& operator>>(istream& In, Money& M); //22
friend ostream& operator<<(ostream& Out, const Money& M); //23

private: //24
    Sign Sgn; //25
    unsigned int Value; //26
}; //27
```

9. [5 points] Using the Line numbers at the right in the Money class declaration above, identify any lines that would cause a compilation error. For each line identified as causing a compilation error very briefly, in one sentence only, explain why the compilation error has occurred and how the code could be corrected to eliminate the error.

   Line 8 (or Line 4 / 5)   Identical signature conflict due to possible omission of parameters of the default argument values. To correct, eliminate the default argument values.
10. [10 points] Assuming all of the member functions of the above class \texttt{Money} have been implemented correctly and that any compilation errors, (if any), in the class declaration are fixed, circle any of the following expressions that would not compile.

\begin{verbatim}
Money M0, M1(100), M2(-200), M3(3.21), M4(4, 0), M5(5000);
M0 = M1 + +M2;
\text{M3 \* 2.0 results in a \texttt{Money} object result, which is multiplied by \texttt{M4}, but the class does not contain a \texttt{Money \* Money} operator overload.}
M0 = M3 \* 2.0 \* M4;
M3 \* 2.0 results in a \texttt{Money} object result, which is multiplied by \texttt{M4}, but the class does not contain a \texttt{Money \* Money} operator overload.
M0 = 3.0 \* M4 - -M2;
The \texttt{Money} class interface supports a \texttt{Money / int} operator overload, but it does not provide for a commutative form.
M0 = 25 / M5;
\end{verbatim}

For the next two questions, consider the following interface variation for the \texttt{Money} class:

\begin{verbatim}
enum Sign {NONNEGATIVE, NEGATIVE, SIGNUNKNOWN};
class Money {
public:
    Money();
    Money(int TotalCents);
    Money(unsigned int TotalCents);
    Money(double Amount);
    Money(int Dollars=0, int Cents=0);
    ~Money();
    //_______________________________; //copy constructor
    //_______________________________; //assignment operator
    //the prototypes of the remaining operators of the class are unchanged from previous \texttt{Money} class declaration (see page 5)

private:
    int* cents; //dynamic integer array
    int numDigits; //number of digits of the Money Value
};
\end{verbatim}

Assume that any compilation errors, (if any), from the previous question that are still present in the class declaration will be fixed later. Note that the underlying implementation representation has changed to an integer array that will dynamically grow and shrink as necessary. The sign will be indicated by a positive/negative value for the first cell value. Each cell value will store nine digits of the monetary value. Thus any monetary object value requiring \leq 9 digits will be stored in an integer array of dimension one.
11. [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, indicated between lines 8 & 9 and 9 & 10 respectively. If you do not believe they are necessary briefly, explain why.

   Money(const Money& source); //copy constructor
   Money& operator=(const Money& source); //assignment operator overload

12. [10 points] Give the implementation for the assignment operator for the above Money class variation, (implement it whether you believe it is necessary or not for the logically correct operation of the class).

   Money& operator=(const Money& source) {
      if (this != &source) { //self assignment check
         if (cents != NULL) //clean up existing memory
            delete [] cents;

         //undefined source check
         if ((source.numDigits == 0) || (source.cents == NULL)) {
            numDigits = 0;
            cents = NULL;
            return(*this);
         } //end if

      numDigits = source.numDigits; //static copy
      cents = new int[(numDigits + 8)/9]; //memory allocation for copy

      for (int i = 0; i < ((numDigits + 8)/9); i++) //deep copy
         cents[i] = source.cents[i];
      } //end if

      return(*this); //cascade enabling
   } //end operator=