1. (18 points) There are *at least* six errors or omissions in the following interface file. Find six errors and write the the line number and correction for each in the space provided below.

```
 1 #ifndef COMPLEX_H
 2
 3
 4 #include <iostream>
 5 using namespace standard;
 6
 7 class Complex {
 8
 9  public:
10
11   /* Constructors */
12   Complex();
13   Complex(double real, double imaginary);
14
15   /* Accessors */
16   double GetReal() const;
17   int GetImaginary() const;
18
19   /* Mutators */
20   void SetReal(double real) const;
21   void SetImaginary(double imaginary);
22
23   /* Overloaded operators */
24   const Complex operator-(const Complex& z);
25   const Complex operator+(const Complex& x, const Complex& y);
26   ostream& operator<<(ostream& sout, const Complex& z);
27
28  private
29
30   double m_real;
31   double m_imaginary;
32
33 }
34
35 #endif
```

Line Number	Correction
2	#define COMPLEX_H
5	using namespace **std**;
17	**double** GetImaginary() const;
20	Remove **const**
25	Delete one of the parameters since part of class.
26	Missing **friend**

Line 28 Need colon (:) after private.

2. (12 points) Complete the code:

a. I want to append the value of the double variable `avg` to the double vector `scores`:

```
scores. push_back(avg) ;
```

b. I want to call the `Fiction()` function of the `Pulp` object pointed to by `pPtr`:

```
Pulp *pPtr = new Pulp;

pPtr -> Fiction();
```

c. The variable `ratings` is a pointer to a double; it points to a dynamically allocated array of doubles. That is,

```
double *ratings = new double[arrayLen];
```

where `arrayLen` is a variable that depends on user input. I want to delete the array:

```
delete [] ratings;
```

d. I am overloading the assignment operator. I need to be sure I handle self-assignment (e.g. `x = x`) properly and that I return the appropriate value:

```
MyClass& MyClass::operator=(const MyClass& rhs) {

  if ( this != &rhs) {

    /* execute only if NOT self-assignment */

  }

  return *this ;

}
```

e. I am writing the interface file for class `Base` and want any class derived from it to have direct access to `Base`'s class variables. Besides `Base` and classes derived from it, other classes should *not* have direct access to the variables:

```
class Base {

    protected:

          int m_classInt;

          string m_classString;
```

3. (8 points) The class `MyArray` has two private class variables, defined in `MyArray.h`:

```
double *m_data;

int m_size;
```

The following constructor is defined in `MyArray.cpp`:

```
MyArray MyArray::MyArray(int size) {
  if (size > 0) {
    m_data = new double[size];
    m_size = size;
  } else {
    m_data = NULL;
    m_size = 0;
  }
}
```

a. Explain why the programmer should also define a copy constructor rather than relying on the default copy constructor provided by the compiler.

The default copy constructor will not copy the elements of the array m_data because it is dynamically allocated: the copy will end-up pointing to the same array as the original.

b. Complete the implementation of the `MyArray` destructor:

```
MyArray::~MyArray() {
    if (m_data != NULL) {

       delete [] m_data;

       m_data = NULL;

       m_size = 0;

    }
}
```

4. (12 points) True or False?

a. $\boxed{\textbf{False}}$ The data members of a `struct` are accessed using the "`*`" operator.

b. $\boxed{\textbf{True}}$ Inheritance implements the "is a" relationship.

c. $\boxed{\textbf{False}}$ The value of a static class variable can not be changed.

d. $\boxed{\textbf{True}}$ *Redefining* (or *overriding*) is when a derived class implements a function with the same signature (name and parameter types) as a function in the base class.

e. $\boxed{\textbf{False}}$ Overloaded operators must always return a `const` value.

f. $\boxed{\textbf{True}}$ When a derived class object is destroyed, the derived class destructor is called before the base class destructor.

g. $\boxed{\textbf{False}}$ A base class object can call a public member function of a derived class.

h. $\boxed{\textbf{True}}$ *Encapsulation* is the the hiding of class variables and function implementations from the user of a class, allowing only controlled access to class data.

i. $\boxed{\textbf{True}}$ A `struct` may be used as a function argument.

j. $\boxed{\textbf{False}}$ *Overloading* implements the "was a" relationship.

k. $\boxed{\textbf{False}}$ The capacity of a vector is always less than or equal to its size.

l. $\boxed{\textbf{True}}$ A `const` member function can be called on a `const` or non-`const` object.

5. (8 points) Consider the following program consisting of the classes `Animal` and `Lion` and a `main()` function:

```
 1 #include <iostream>
 2 using namespace std;
 3 
 4 class Animal {
 5 public:
 6   void Eats() { cout << "Eats food." << endl; }
 7 };
 8 
 9 class Lion : public Animal {
10 public:
11   Lion() : Animal(), m_name("Leo") {}
12   Lion(string name) : Animal(), m_name(name) {}
13   void Eats() { cout << m_name << " eats meat." << endl; }
14   void Sleep() { cout << "Ahh...a nice nap!" << endl; }
15 private:
16   string m_name;
17 };
18 
19 int main() {
20   Animal animal;
21   Lion lion;
22 
23   animal.Eats();
24   lion.Eats();
25 
26   animal.Sleep();
27 
28   return 0;
29 }
```

a. Line 26 causes an error when the program is compiled. Why?

Sleep() is a method of the derived class, but animal is a base-class object. Base-class objects are unable to call derived-class functions.

b. If Line 26 is deleted and the program is compiled and run, what output will it produce?

"Eats food."

"Leo eats meat."

6. (12 points) A program needs to create a dynamically-allocated two-dimensional array. The variables `nrows` and `ncols` contain the required number of rows and columns, respectively. Complete the code to create the `nrows`-by-`ncols` integer matrix `intArray` and initialize its elements to zero:

```
int **intArray = new int*[nrows];
for (int i = 0; i < nrows; i++) {
  intArray[i] = new int[ncols];
  for (int j = 0; j < ncols; j++)
    intArray[i][j] = 0;
}
```

7. Consider the following interface (.h) file for a `Vehicle` class:

```
#ifndef VEHICLE_H
#define VEHICLE_H

class Vehicle {
 public:

  /* Default Constructor - creates a vehicle that can carry
     passengers AND freight.                                 */

  Vehicle();

  /* Non-default Constructor - select whether vehicle can
     carry passengers and/or freight.                      */

  Vehicle(bool takesPassengers, bool takesFreight);

 private:
  bool m_takesPassengers;   // true if vehicle can carry passengers
  bool m_takesFreight;      // true if vehicle can carry freight
};

#endif
```

The `Car` class is to be derived from the `Vehicle` class. A `Car` can carry passengers, but can *not* carry freight. `Car` contains three additional private class variables: an integer `m_numSeats` indicating how many seats the car has, an integer `m_seatsAvailable` indicating how many seats are available, and a string array `m_passengers` containing the names of the passengers in the car.

a. (5 points) Write the implementation of a default constructor which creates a `Car` with five seats. Initially, all the seats should be available. The passenger array must be dynamically allocated and be of the appropriate size (one element per seat).

```
Car::Car() : Vehicle(true, false), m_numSeats(5), m_seatsAvailable(5) {
    m_passengers = new string[5];
}
```

b. (5 points) Write the implementation of a non-default constructor that creates a `Car` with the number of seats specified as an argument. As with the default constructor, initially all seats should be available and the passenger array should be dynamically allocated and be of the appropriate size (one element per seat).

```
Car::Car(int numSeats) : Vehicle(true, false), m_numSeats(numSeats),
          m_seatsAvailable(numSeats) {
    m_passengers = new string[numSeats];
}
```

c. (5 points) Write the implementation of a `Car` destructor.

```
Car::~Car() {
    delete [] m_passengers;
}
```

d. (5 points) If a seat is available, the function `AddPassenger(string name)` adds `name` to the passenger array and decrements the number of seats available. If no seats are available, the function prints a warning message. Write the implementation of the function:

```
void Car::AddPassenger(string name) {
   if ( m_seatsAvailable > 0 ) {
      m_passengers[m_numSeats - m_seatsAvailable] = name;
       m_seatsAvailable--;
   } else
       cerr << "Sorry, the car is full! << endl;
}
```

e. (10 points) Write the *complete* interface (`.h`) file for the `Car` class, including both constructors, the destructor, and `AddPassenger()`. Do **not** include comments.

```
#ifndef CAR_H
#define CAR_H
using namespace std;
class Car : public Vehicle {
  public:
    Car();
    Car(int numSeats);
    void AddPassenger(string name);
    ~Car();
  private:
    int m_numSeats;
    int m_seatsAvailable;
    string* m_passengers;
};
#endif
```