1. (15 points) There are *at least* six errors or omissions in the following class definition. Find five errors and write the the line numbers and corrections in the space provided below.

```
 1 class Train {
 2  public:
 3   Train() : m_cars = NULL, m_numCars = 0, m_diesel = false {}
 4
 5   Train(TrainCar *cars, int numCars)
 6    : m_numCars(numCars), m_diesel(false) {
 7     m_cars = new TrainCar[numCars];
 8     for (int i = 0; i < numCars; i++)
 9       m_cars[i] = cars[i];
10   }
11
12   Train(const Train &t)
13    : m_numCars(t.m_numCars), m_diesel(t.m_diesel) {
14     m_cars = t.m_cars;
15   }
16
17   ~Train() {
18     delete m_cars;
19   }
20
21   int setDiesel() const {
22     m_diesel = true;
23   }
24
25   ostream& operator<<(ostream& sout, const Train& t) {
26     cout << "The train has " << t.m_length << " cars.";
27     return sout;
28   }
29
30  private:
31   Car *m_cars;
32   int m_numCars;
33   bool m_diesel;
34 };
```

Line Number	Correction
3	m_cars(NULL), m_numCars(0), m_diesel(false)
14	Replace shallow copy of t.m_cars with deep copy
18	delete [] m_cars
21	change int to void
21	delete const
25	add friend
26	change cout to sout
26	change t.m_length to t.m_numCars

2. (24 points) Complete the code: **3 points each**

a. I want to append the value of the int variable `numCars` to the int vector `trains`:

```
trains. push_back(numCars) ;
```

b. I want to call the `Abides()` function of the `Dude` object pointed to by `dPtr`:

```
Dude *dPtr = new Dude;

dPtr -> Abides();
```

c. The function `buggy()` may throw an exception of type `BuggyDataEx`, which has a `what()` function. I want my code to handle the exception should it occur:

```
try {
   buggy();

} catch ( BuggyDataEx &ex ) {
   out << ex.what() << endl;
   // handle the exception
}
```

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:

```
Train& Train::operator=(const Train& t) {

  if (this != &t ) {

    // execute only if NOT self-assignment

  }

  return *this ;

}
```

e. I am writing the `FreightTrain` class which is derived from the `Train` class:

```
class FreightTrain : public Train {
  // class declaration goes here
};
```

3. The class `Car` has two private class variables, defined in `Car.h`:

```
Seat *m_seats;
unsigned int m_numSeats;
```

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

```
Car::Car(unsigned int numSeats) : m_numSeats(numSeats) {
  if (numSeats == 0)
    m_seats = NULL;
  else
    m_seats = new Seat[m_numSeats];
}
```

a. (5 points) Why should the programmer define a copy constructor rather than rely on the default copy constructor provided by the compiler?

The default copy constructor provides a **shallow copy** and will not copy the m_seats array. The programmer must write a **deep copy** constructor.

(12 points) Complete the implementation of the `Car` assignment operator:

```
Car& Car::operator=(const Car& c) {

  if (this != &c) {

    if (m_seats != NULL) {

      delete [] m_seats;
      m_seats = NULL;

    }

    if (c.m_numSeats > 0 {

      m_seats = new Seat[c.m_numSeats];

      for (int i = 0; i < c.m_numSeats;i++)
        m_seats[i] = c.m_seats[i];

    }
    m_numSeats = c.m_numSeats;
  }
  return *this;
}
```

4. (14 points) True or False?

a. **True** The data members of an object are accessed using the "`.`" operator.

b. **False** *Overloading* implements the "was a" relationship.

c. **True** A derived class object can call a protected member function of a base class.

d. **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. **False** Overloaded operators must never return a `const` value.

f. **False** When a derived class object is destroyed, the base class destructor is called before the derived class destructor.

g. **True** An object may be used as the return value of a function.

h. **False** A *shallow copy* will copy data in dynamically allocated arrays so long as the arrays aren't too long.

i. **False** Exceptions allow low-level code to handle errors so that high-level code doesn't have to.

j. **True** Inheritance implements the "is a" relationship.

k. **False** Elements of a vector can only be accessed using the `at()` function.

l. **True** A `const` member function can be called on a `const` or non-`const` object.

m. **True** A `friend` function can access the private functions and variables of the class.

n. **True** For every `new` there should be a `delete`.

5. (10 points) Consider the following program consisting of the classes `Vehicle` and `Tractor` and a `main()` function:

```
 1 #include <iostream>
 2 using namespace std;
 3
 4 class Vehicle {
 5 public:
 6   void move(){ cout << "The vehicle is moving." << endl; }
 7 };
 8
 9 class Tractor : public Vehicle {
10  public:
11   Tractor() : Vehicle(), m_make("John Deere"){}
12   Tractor(string make) : Vehicle(), m_make(make){}
13   void move(){ cout << m_make << " tractor is moving." << endl; }
14   void plow(){
15     cout << m_make << " tractor is plowing the field." << endl; }
16  private:
17   string m_make;
18 };
19
20 int main() {
21   Vehicle vehicle;
22   Tractor tractor("Massey-Ferguson");
23
24   vehicle.move();
25   tractor.move();
26
27   vehicle.plow();
28
29   return 0;
30 }
```

a. Line 27 causes an error when the program is compiled. Why?

plow() is a method of the derived class; it can not be called on a Vehicle object.

b. If Line 27 is deleted and the program is compiled and run, what output will it produce?

The vehicle is moving.

Massey-Ferguson tractor is moving.

6. A linked list is used to store integers in increasing order. The nodes of the linked list have two public variables: `int m_value` and `Node *m_next`. The first node of the list is a "dummy node" and the pointer variable `m_head` points to the dummy node.

a. (12 points) The program must insert a new node with value `val` into the list:

```
Node *current = m_head;
while( current->m_next != NULL ) {
  if(current->m_next->m_value > val) {
    Node* ptr = new Node(val);
    ptr->m_next = current->m_next;
    current->m_next = ptr;
    return;
  }
  current = current->m_next;
}
current->next = new Node(val);
```

b. (8 points) The program must remove all nodes with a given value `val`:

```
Node *current = m_head;
while(current->m_next != NULL ) {
  if( current->m_next->m_value == val) {
    Node *ptr = current->m_next;
    current->m_next = current->m_next->m_next;
    delete ptr;
  }
  current = current->m_next;
}
```