HW3 Solution

Problem 1:

Since all Vehicles have an unique id, and since the way you find out the id is going to be the same for all kinds of vehicles, it should be a data member of Vehicle, and the function to retrieve it need not be virtual.

The different kinds of vehicles may have different behavior for the description and canHover functions, so these should be virtual. Since most types of vehicles can hover, it's convenient to have Vehicle::canHover have an implementation (returning true) that derived classes may inherit if they wish. There is no reasonable default behavior for the description, so this should be pure virtual.

Observe how the constructors for the derived classes pass the id to the Vehicle constructor. Examine Balloon's constructor especially. Also note that Vehicle, because it's designed as a base class, has a virtual destructor. Observe what happens if its destructor were not declared virtual.

```
// ======================================== Vehicle
class Vehicle
{
  public:
    Vehicle(string ident);
    string id() const;
    virtual string description() const = 0;
    virtual bool canHover() const;
    virtual ~Vehicle() {}
  private:
    string m_id;
};

Vehicle::Vehicle(string ident)
 : m_id(ident)
{}

string Vehicle::id() const
{
  return m_id;
}

bool Vehicle::canHover() const
{
  return true;
}

// ======================================== Drone
class Drone : public Vehicle
{
  public:
    Drone(string ident);
    virtual string description() const;
```

```
    virtual ~Drone();
};

Drone::Drone(string ident)
 : Vehicle(ident)
{}

string Drone::description() const
{
  return "a drone";
}

Drone::~Drone()
{
  cout << "Destroying " << id() << ", a drone" << endl;
}

// ====================================== Balloon
class Balloon : public Vehicle
{
  public:
    Balloon(string ident, double diameter);
    virtual string description() const;
    virtual ~Balloon();
  private:
    double m_diameter;
};

Balloon::Balloon(string ident, double diameter)
 : Vehicle(ident), m_diameter(diameter)
{}

string Balloon::description() const
{
  if (m_diameter >= 8)
    return "a large balloon";
  else
    return "a small balloon";
}

Balloon::~Balloon()
{
  cout << "Destroying the balloon " << id() << endl;
}

// ====================================== Satellite
class Satellite : public Vehicle
{
  public:
    Satellite(string ident);
    virtual string description() const;
    virtual bool canHover() const;
    virtual ~Satellite();
};
```

```
Satellite::Satellite(string ident)
 : Vehicle(ident)
{}

string Satellite::description() const
{
  return "a satellite";
}

bool Satellite::canHover() const
{
  return true;
}

Satellite::~Satellite()
{
  cout << "Destroying the satellite " + id() << endl;
}
```

Problem 2:

```
  // Return true if the somePredicate function returns true for at
  // least one of the array elements; return false otherwise.
bool anyTrue(const double a[], int n)
{
    if (n <= 0)
        return false;
    if (somePredicate(a[0]))
        return true;
    return anyTrue(a+1, n-1);
}

  // Return the number of elements in the array for which the
  // somePredicate function returns true.
int countTrue(const double a[], int n)
{
    if (n <= 0)
        return 0;
    int t = somePredicate(a[0]);  // 1 if true, 0 if false
    return t + countTrue(a+1, n-1);
}

  // Return the subscript of the first element in the array for which
  // the somePredicate function returns true.  If there is no such
  // element, return -1.
int firstTrue(const double a[], int n)
{
    if (n <= 0)
        return -1;
```

```
    if (somePredicate(a[0]))
        return 0;
    int k = firstTrue(a+1, n-1);
    if (k == -1)
        return -1;
    return 1 + k;  // element k of "the rest of a" is element 1+k of a
}

  // Return the subscript of the smallest element in the array (i.e.,
  // return the smallest subscript m such that a[m] <= a[k] for all
  // k from 0 to n-1).  If the function is told that no doubles are to
  // be considered in the array, return -1.
int indexOfMinimum(const double a[], int n)
{
    if (n <= 0)
        return -1;
    if (n == 1)
        return 0;
    int k = 1 + indexOfMinimum(a+1, n-1);  // indexOfMinimum can't return -1 here
    return a[0] <= a[k] ? 0 : k;

    // Here's an alternative for the last two lines above:
    // int k = indexOfMinimum(a, n-1);  // indexOfMinimum can't return -1 here
    // return a[k] <= a[n-1] ? k : n-1;
}

  // If all n1 elements of a1 appear in the n2 element array a2, in
  // the same order (though not necessarily consecutively), then
  // return true; otherwise (i.e., if the array a1 is not a
  // not-necessarily-contiguous subsequence of a2), return false.
  // (Of course, if a1 is empty (i.e., n1 is 0), return true.)
  // For example, if a2 is the 7 element array
  //    10 50 40 20 50 40 30
  // then the function should return true if a1 is
  //    50 20 30
  // or
  //    50 40 40
  // and it should return false if a1 is
  //    50 30 20
  // or
  //    10 20 20
bool isIn(const double a1[], int n1, const double a2[], int n2)
{
    if (n1 <= 0)
        return true;
    if (n2 < n1)
        return false;

      // If we get here, a1 and a2 are nonempty
    if (a1[0] == a2[0])
        return isIn(a1+1, n1-1, a2+1, n2-1);  // rest of a1, rest of a2
    else
        return isIn(a1, n1, a2+1, n2-1);  // all of a1, rest of a2
}
```

Problem 3:

```
bool pathExists(string maze[], int nRows, int nCols, int sr, int sc, int er, int
ec)
{
    if (sr == er  &&  sc == ec)
        return true;

    maze[sr][sc] = '@';  // anything non-'.' will do

    if (maze[sr][sc+1] == '.'  &&  pathExists(maze, nRows, nCols, sr, sc+1, er,
ec))
        return true;
    if (maze[sr-1][sc] == '.'  &&  pathExists(maze, nRows, nCols, sr-1, sc, er,
ec))
        return true;
    if (maze[sr][sc-1] == '.'  &&  pathExists(maze, nRows, nCols, sr, sc-1, er,
ec))
        return true;
    if (maze[sr+1][sc] == '.'  &&  pathExists(maze, nRows, nCols, sr+1, sc, er,
ec))
        return true;

    return false;
}
```

or

```
bool pathExists(string maze[], int nRows, int nCols, int sr, int sc, int er, int
ec)
{
    if (maze[sr][sc] != '.')
        return false;

    if (sr == er  &&  sc == ec)
        return true;

    maze[sr][sc] = '@';  // anything non-'.' will do

    if (pathExists(maze, nRows, nCols, sr, sc+1, er, ec))
        return true;
    if (pathExists(maze, nRows, nCols, sr-1, sc, er, ec))
        return true;
    if (pathExists(maze, nRows, nCols, sr, sc-1, er, ec))
        return true;
    if (pathExists(maze, nRows, nCols, sr+1, sc, er, ec))
        return true;
```

```
    return false;
}
```

Problem 4:

```
  // Return the number of ways that all n1 elements of a1 appear in
  // the n2 element array a2 in the same order (though not necessarily
  // consecutively).  We decree that he empty sequence (i.e. one where
  // n1 is 0) appears in a sequence of length n2 in 1 way, even if n2
  // is 0.  For example, if a2 is the 7 element array
  //    10 50 40 20 50 40 30
  // then for this value of a1     the function must return
  //    10 20 40                        1
  //    10 40 30                        2
  //    20 10 40                        0
  //    50 40 30                        3
int countIsIn(const double a1[], int n1, const double a2[], int n2)
{
    if (n1 <= 0)
        return 1;
    if (n2 < n1)
        return 0;

      // If we get here, a1 and a2 are nonempty
    int t = countIsIn(a1, n1, a2+1, n2-1);  // all of a1, rest of a2
    if (a1[0] == a2[0])
        t += countIsIn(a1+1, n1-1, a2+1, n2-1);  // rest of a1, rest of a2
    return t;
}

  // Exchange two doubles
void exchange(double& x, double& y)
{
    double t = x;
    x = y;
    y = t;
}

  // Rearrange the elements of the array so that all the elements
  // whose value is > divider come before all the other elements,
  // and all the elements whose value is < divider come after all
  // the other elements.  Upon return, firstNotGreater is set to the
  // index of the first element in the rearranged array that is
  // <= divider, or n if there is no such element, and firstLess is
  // set to the index of the first element that is < divider, or n
  // if there is no such element.
  // In other words, upon return from the function, the array is a
  // permutation of its original value such that
  //   * for 0 <= i < firstNotGreater, a[i] > divider
```

```
  //   * for firstNotGreater <= i < firstLess, a[i] == divider
  //   * for firstLess <= i < n, a[i] < divider
  // All the elements > divider end up in no particular order.
  // All the elements < divider end up in no particular order.
void divide(double a[], int n, double divider,
                                    int& firstNotGreater, int& firstLess)
{
    if (n < 0)
        n = 0;

      // It will always be the case that just before evaluating the loop
      // condition:
      //  firstNotGreater <= firstUnknown and firstUnknown <= firstLess
      //  Every element earlier than position firstNotGreater is > divider
      //  Every element from position firstNotGreater to firstUnknown-1 is
      //    == divider
      //  Every element from firstUnknown to firstLess-1 is not known yet
      //  Every element at position firstLess or later is < divider

    firstNotGreater = 0;
    firstLess = n;
    int firstUnknown = 0;
    while (firstUnknown < firstLess)
    {
        if (a[firstUnknown] < divider)
        {
            firstLess--;
            exchange(a[firstUnknown], a[firstLess]);
        }
        else
        {
            if (a[firstUnknown] > divider)
            {
                exchange(a[firstNotGreater], a[firstUnknown]);
                firstNotGreater++;
            }
            firstUnknown++;
        }
    }
}

  // Rearrange the elements of the array so that
  // a[0] >= a[1] >= a[2] >= ... >= a[n-2] >= a[n-1]
  // If n <= 1, do nothing.
void order(double a[], int n)
{
    if (n <= 1)
        return;

      // Split using a[0] as the divider (any element would do).
    int firstNotGreater;
    int firstLess;
    divide(a, n, a[0], firstNotGreater, firstLess);
```

```
        // sort the elements > divider
      order(a, firstNotGreater);

        // sort the elements < divider
      order(a+firstLess, n-firstLess);
  }
```