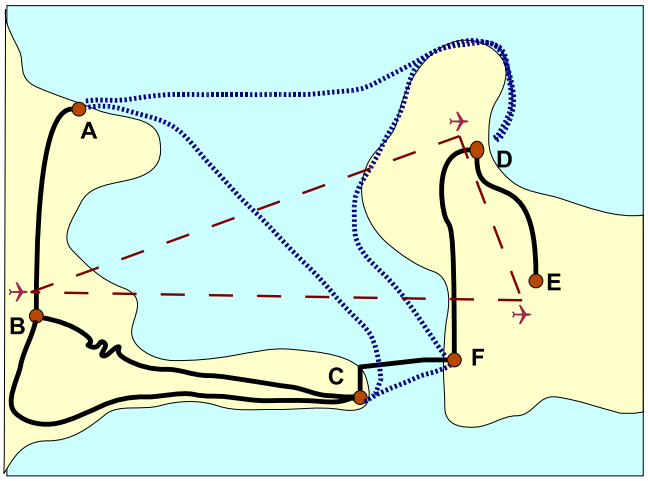
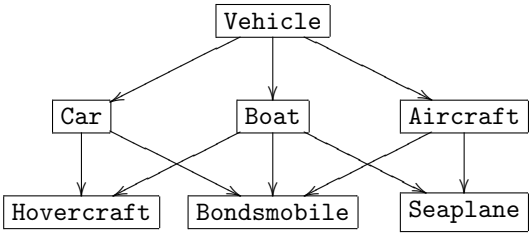
CEng 242 Homework 5

Due 17^{th} May 2004

In this homework you will implement a simple application to experiment multiple inheritance and virtual functions.
You are given a group of cities connected via three kinds of transportation: land, sea and air.



Each transportation link is given with its distance. You are given different sorts of vehicles to travel between these cities. These vehicles may have different speeds and features allowing them to travel on different ways. All vehicles can be derived from the same superclass. `Vehicle` class hierarchy is given as:



`Car` denotes a derived class of vehicles which can travel on land using the road links. Similarly a `Boat` can travel on sea and `Aircraft` can fly in predefined air routes and land on airports. The vehicles `Hovercraft` and `Seaplane` combines features of two classes. Finally the `Bondsmobile` is a futuristic vehicle designed for spies and travel on land, sea and flies.

The `Vehicle` class and some other class definitions are given as follows:

```
enum Optimization { CHEAPEST, SHORTEST };
struct Link {
    char source[30], destination[30];
    enum LinkType { LAND, WATER, AIR, NOLINK } type;
    double hours,cost;
};
class Vehicle {
    char name[50];
public:
    Vehicle(const char *);
    List<Link> findOptimumPath(const char *, const char *, Optimization);
```

```
    virtual Link getOptLink(const char *, const char *, Optimization)=0;
    static class Map {...} map;
};
class Car: virtual public Vehicle {
    double costperkm, kmperhour;
public:
    Car(const char *, double, double);
    virtual Link getOptLink(const char *, const char *, Optimization);
};
class Boat:virtual public Vehicle {
    double costperkm, kmperhour;
public:
    Boat(const char *, double, double);
    virtual Link getOptLink(const char *, const char *, Optimization);
};
class Aircraft:virtual public Vehicle {
    double costperkm, kmperhour;
public:
    Aircraft(const char *, double, double);
    virtual Link getOptLink(const char *, const char *, Optimization);
};
class Hovercraft:public Car, public Boat {
public:
    Hovercraft(const char *, double, double, double, double);
    virtual Link getOptLink(const char *, const char *, Optimization);
};
class Bondsmobile:public Car, public Boat, public Aircraft {
public:
    Bondsmobile(const char *, double, double, double, double, double, double);
    virtual Link getOptLink(const char *, const char *, Optimization);
};
class Seaplane:public Boat, public Aircraft {
public:
    Seaplane(const char *, double, double, double, double);
    virtual Link getOptLink(const char *, const char *, Optimization);
};
```

You will be given the cities, the possible direct links between the cities, and their distances as input. You will keep all these information in a global `Map` object.
The members of `Vehicle` and their meanings are:

Constructors First parameter is always the name of the vehicle. Then comes the values of `costperkm` and `kmperhour` members are given. If the class is derived from two or more classes these two parameters repeated for the superclasses in the inheritance specification order.

When this vehicle is to follow a link, you will calculate the cost of the travel and the hours required to arrive the destination by using these two members and the distance between two cities.

`Link getOptLink(const char *, const char *, Optimization)`
Depending on the type of the optimization, it tries to find the minimum cost, or the fastest direct connection between two cities. First two parameters are the name of the source and destination cities respectively, and the third parameter is the optimization type.

This function returns a `Link` structure containing the link information, city names, type of the link, the hours and cost required to follow the link. Depending on the vehicles capabilities, the class may choose among the different alternatives. For example between the cities D and F, both sea and land connection exists and a `Hovercraft` may choose the best of the links, the cheapest or the shortest (in terms of time).

Note that this virtual function is not implemented in `Vehicle` class but in the derived classes.

`List<Link> findOptimumPath(const char *,const char *,Optimization)`
This function should **only** be implemented in `Vehicle` class. It calls `getOptLink(...)` implementation of the subclasses and tries to find the optimum path among two cities. The first two parameters are the names of the source and destination cities respectively, and the third parameter is the type of the optimization, the cheapest path or the shortest (fastest) path.

This function should return a list of `Link` typed values, that will keep the list of intermediate cities and links to follow during the travel. The optimization algorithm is explained in following sections.

`class Map {...} map`
This class member keeps the global map information. All city/link information is inserted in this member and all vehicles access it to find the optimum link among two cities.

The class should be defined inside of the `Vehicle` class definition.

You can implement the `List<...>` class in any way you want as long as you hide the member variables and respect OO programming spirit. Provide these two functions:

`Link Link::operator+(const Link &)`
Adds two `Link` structure if they are connected. For example if the first link is from city A to C and the second link is from city C to city D, resulting link will be from city A to city D. The `hours` and `cost` fields are summed. The `type` field will be same as the first operands type.

If the links are not connected, the resulting type will be `NOLINK` indicating this link is invalid. Similarly the addition of any `NOLINK` type operand will result in `NOLINK` type.

`T List<T>::sum()`
Returns the sume of all members in the list with '+' operator.

`int List<T>::isempty()`
Returns non zero if the list is not empty. If the list is empty, returns 0.

`ostream & operator <<(ostream &, const Link &)`
Output a single line of link information as one of:

```
Source          Destination   AIR     3.20hrs   15.5MTL
Source          Destination   LAND    2.00hrs   10.0MTL
Source          Destination   SEA     5.00hrs    1.0MTL
```

Use the format as the following `printf()` generates:
`printf("%-20s %-20s %-5s %6.2fhrs %8.1fMTL\n",src,dst,type,hours,cost);`

`ostream & operator <<(ostream &, const List<T> &)`
Output the elements of the list in the order of appearance in the list.

Similarly, you are free to design and implement the `Map` class which will keep the length and types of the links between the cities. The number of cities has an upper limit `50`. Typically adjacency matrix representation is used in such applications. The $M(i, j)$ location

in this matrix keeps the information of the link between the cities i and j. The links are bidirectional, that means $M(i,j) = M(j,i)$ holds for all cities. For the interface of this class (actually `Vehicle::Map`) you should implement the following member functions:

`void Map::addCity(const char *)`
Inserts a new city to the map. If the city already exists, silently ignored. Initially city has no connection to other cities.

`void Map::addLink(const char *, const char *, Link::LinkType, int)`
`addLink(`*`src`*`, *`*`dst`*`, `*`type`*`, `*`distance`*`)` Inserts a new direct link between the cities *src* and *dst*. This link will be bidirectional. The type of the link (LAND, SEA, AIR) and the distance (km's) is given in third and fourth parameters. Note that there may be any number of direct links between two cities with duplicate types. Like two land links, one is cheaper and one is shorter may exist (see B and C in the figure).

Sample main code would be:

```
#include<iostream.h>
#include"hw5.h"
int main() {
      int ncities;
      char src[30], dst[30];
      int t;
      double d;
      Link::LinkType typ;

      cout << "Enter the number of cities";
      cin >> ncities;
      for (int i=0;i<ncities;i++) {
            cout << "City Name:";
            cin >> src;
            Vehicle::map.addCity(src);
      }
      cout << "Would you like to add a link?"; cin << src;
      while ( src[0]=='Y' || src[0]=='y') {
            Link newlink;
            cout << "Source city:"; cin >> src;
            cout << "Dest city:"; cin >> dst;
            cout << "distance (km):"; cin >> d;
            cout << "type (1:Air, 2:Water, *:Land):"; cin >> t;
            swich (t) {
            case 1: typ=AIR;
                  break;
            case 2: typ=WATER;
                  break;
            default: typ=LAND
            }
            Vehicle::map.addLink(src,dst,typ,d);
            cout << "Would you like to add a link?"; cin << src;
      }
      int nvehicles;
      cout << "Enter the number of vehicles:"; cin >> nvehicles;
      Vehicle *vehicles=new Vehicle *[nvehicles];

      // Read names, types and attributes of the vehicles
      // and construct them. *(vehicles[i]) gives you a vehicle
```

```
      // of any kind. Insert code here .....

      cout << "Ask an optimum path?:" ; cin << src;
      while ( src[0]=='Y' || src[0]=='y') {
            List<Link> opath;
    Link sum;
            cout << "Source city:"; cin >> src;
            cout << "Dest city:"; cin >> dst;
            cout << "Vehicle id (0<=i<nvehicles):"; cin >> t;
            opath=vehicles[t]->findOptimumPath(src,dst,CHEAPEST);
            if (!opath.isempty()) {
                  sum=opath.sum();
                  cout << "Cheapest path takes " << sum.cost
                        << "MTL and " << sum.hours << "hours\n";
                  cout << "Path is given as:\n-----------------\n"
                        << opath;
            }
            opath=vehicles[t]->findOptimumPath(src,dst,SHORTEST);
            if (!opath.isempty()) {
                  sum=opath.sum();
                  cout << "Cheapest path takes " << sum.cost
                        << "MTL and " << sum.hours << "hours\n";
                  cout << "Path is given as:\n-----------------\n"
                        << opath;
            }
            cout << "Ask an optimum path?"; cin << src;
      }
}
```

Similar to the previous homework, put only class prototypes (no inline definition) and required type definitions in hw5.h, include this file in hw5.cpp and in your for yourself only main program code hw5main.cpp. Your codes should be compiled as:

```
g++ -c hw5.cpp
g++ -o hw5main hw5main.cpp hw5.o
```

Shortest Path

Shortest path algorithm is a simple incremental algorithm and the approach is to find shortest path of all cities starting from the source city. The cities are added with the order of closeness to the source city.

Let $Dist(i)$ denote the direct or indirect distance of city i from the source city. Initially this structure only contains the direct links from the source city. All other, not directly connected, cities are assumed to have infinite distance. Let $Link(i, j)$ denote the distance of a direct link from city i to city j.

Let S denote the set of cities where the shortest path calculation is completed. Initially this set only contains the source city. The calculation stops when the destination city is included in this set.

So the algorithm can be written as follows:

Set *Dist(i)=Link(src,i)* for all i
Set S=$\{src\}$
While *dest* city is not a member of S {
    Set u to city which has the minimum *Dist(i)* value where $u \notin S$
    Add u to S
    For each j with u is directly connected; Link(u,j)$\neq \infty$ {

```
            update the Dist(j) as: Dist(j)=min{Dist(j), Dist(u)+Link(u,j)}
        }
    }
```

You need to be careful about the links in your homework. There may be more than one link between two cities and a vehicle can take these links based on its capability. Also SHORTEST depends on the speed of the vehicle and CHEAPEST depends on the cost per kilometer of the vehicle. If there is no path between two cities, you should return an empty list as the path. That means you won't be able to choose a u value as the minimum $Dist(i)$ value, all have infinite distance.
You also need to keep track of the shortest path during the algorithm. So keep the partial path information along with the length of the path in $Dist(i)$.
There are hundreds of web sites explaining this algorithm. See demos at:
`http://carbon.cudenver.edu/~hgreenbe/sessions/dijkstra/DijkstraApplet.html`
`http://www-b2.is.tokushima-u.ac.jp/~ikeda/suuri/dijkstra/DijkstraApp.shtml`

You are not allowed to copy code from Internet and use libraries providing the shortest path solution. You can use Standard Template Library for this homework.