Finding The Cheapest Route, Reading

This is probably what you were hoping for -- a shortest route solution where we *do* take into consideration weighted edges. This is what the maps programs and GPSs use to calculate your route home, based on shortest distance or time or toll cost or whatever.

The first thing we have to face is the storing of another piece of information -- edge cost. A **double** is a good choice for the data type, because it works for most representations of cost. We could use an additional STL list as an attribute in **struct Node**, but the problem with that will be that we traverse the neighbors list with an iterator instead of an index. We cannot easily traverse two lists in tandem like we can traverse two arrays. We could rethink our choice of an STL list, or we can use the list to store an *object* containing an int (the neighbor's index) and a double (the code to get to the neighbor). That would require another struct, *or* we can learn about and use a feature of the C++ STL designed for just this purpose. It's called the "pair".

The STL pair Template

The pair is a very simple data structure of just two values. It's a generic 2-attribute struct. It's in a library called **#include<utility>** and requires **using namespace std;**. A pair object is declared like this:

```
pair<int, double> x; // the data types for the two values are int and double
```

The first attribute is accessed as **x.first** and the second as **x.second** -- easy-to-remember generic attribute names.

We can replace the **list<int> neighbors;** attribute with **list<pair<int, double> > neighbors;**. Note the space separating the two closing angle brackets, because without that, they form a stream extraction operator!

This changes a few things. Traversal now works like this:

```
list<pair<int, double> >::const_iterator it;
for (it = database[i].neighbors.begin(); it != database[i].neighbors.end(); it++)
{
  it->first is a neighboring node's index
  database[it->first] is a neighboring node
  it->second is the cost of getting to that node
}
```

Also the sample code to build a graph from an external flat file changes, because there needs to be a line with *cost*, and it needs to be stored in the database's nodes.

```
#include <fstream>
#include <iostream>
#include <list>
#include <string>
#include <utility>
#include <vector>
using namespace std;

#include <cstdlib> // for atof

struct Node
{
  string name;
  list<pair<int, double> > neighbors;
};
```

```
int main()
{
  ifstream fin;
  fin.open("cities.txt");
  if (!fin.good()) throw "I/O error";

  // process the input file
  vector<Node> database;
  while (fin.good()) // EOF loop
  {
    string fromCity, toCity, cost;

    // read one edge
    getline(fin, fromCity);
    getline(fin, toCity);
    getline(fin, cost); // as a C++ string, to be converted later
    fin.ignore(1000, 10); // skip the separator

    // add nodes for new cities included in the edge
    int iToNode = -1, iFromNode = -1, i;
    for (i = 0; i < database.size(); i++) // seek "to" city
      if (database[i].name == fromCity)
        break;
    if (i == database.size()) // not in database yet
    {
      // store the node if it is new
      Node fromNode = {fromCity};
      database.push_back(fromNode);
    }
    iFromNode = i;

    for (i = 0; i < database.size(); i++) // seek "from" city
      if (database[i].name == toCity)
        break;
    if (i == database.size()) // not in vector yet
    {
      // store the node if it is new
      Node toNode = {toCity};
      database.push_back(toNode);
    }
    iToNode = i;

    // store bi-directional edges
    pair<int, double> edge;
    edge.first = iToNode;
    edge.second = atof(cost.c_str());
    database[iFromNode].neighbors.push_back(edge);
    edge.first = iFromNode;
    database[iToNode].neighbors.push_back(edge);
  }
  fin.close();
  cout << "Input file processed\n\n";
}
```

Note that the database file now has an extra line following each pair of cities -- the cost as a whole number or floating point.