# Chapter 15

# Case study – Containers

In this case study we will illustrate the use of several of the container classes provided as part of the Standard Template Library, which was described earlier in Chapter 9. To do this, we will examine two example problems. The first will be an algorithm for computing shortest paths between pairs of points in a graph. The second example problem will involve scanning a text document, and developing a concordance.

## 15.1 Graph Shortest Path Problem

Imagine that we have a weighted graph that represents, for example, the cost to travel between pairs of cities. The graph is directed, meaning that travel can be made in one direction but not the other. An example graph is shown in Figure 15.1. The task is to determine not only the minimum cost to travel from one city to each of the others, but also the path to follow in making the journey.

To see how we could represent a graph internally, consider first the information we need to maintain for a single city in isolation. If we consider just one city, Phoenix for example, we need to know the names of the cities that can be reached starting from Phoenix, and the cost of each journey. This information could be maintained by a *map*, which is an indexed dictionary structure. The keys in the map will be the destination cities, while the value fields will be the cost:

| | |
|---|---|
| Phoenix: | [ Peoria, 4 ] |
| | [ Pittsburgh, 10 ] |
| | [ Pueblo, 3 ] |

Let us call this information a `cityInfo`. In terms of the STL data structures, this could be represented by a `map` in which the keys are represented by constant character pointers (the common representation for strings in C++) and the value fields by integers. To use the `map`

DEFINE *A map is an indexed collection, similar to a Java Dictionary*

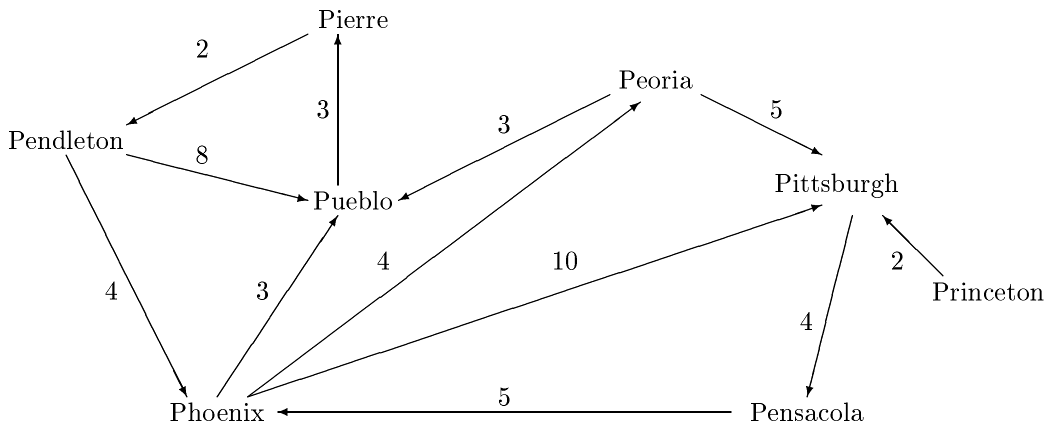


Figure 15.1: A Weighted Graph

data type, we need a *function object* that defines the ordering on keys. Since the default ordering on pointer values (ordering by location in memory) is not appropriate, we must create a new data type. We do this as follows:

```
class charCompare {
public:
    bool operator () (const char * left, const char * right) const
    {
        return strcmp(left, right) < 0;
    }
};
```

An instance of charCompare is a function object, an object that can be used like a function. The behavior we desire is a simulation of the less than operator applied to our key values. When invoked as a function it takes two arguments, which are pointers to character arrays. Using the standard library routine strcmp, it compares the two string texts. A negative value indicates that the first is lexicographically smaller than the second, and hence that the less than operator should return a true value.

Using charCompare, the cityInfo data type could be defined as follows:

```
    typedef map <const char *, unsigned int, charCompare> cityInfo;
```

That is, we declare the name cityInfo to be a synonym for a type of map in which the key field is a character pointer, and the value field is an unsigned integer. The third argument represents the comparison algorithm that will be used to determine the ordering of the keys. We have chosen to use a typedef (see Section 12.3) to declare the new name, rather than defining a new class. This is because all the behavior we need is provided already by the map data type, when properly parameterized. The typedef creates a *synonym* name for the new structure, but does not create any new class structure. We can use this synonym name in our later programs to help simplify the code and make it more readable.

To represent the entire graph, we need only maintain the city information for each city. We can again use a map for this purpose. The key field in the map will once again be a city, and the value field will be a cityInfo that encodes the information associated with the city. Let us use the name graph to represent the entire data structure:

```
typedef map <const char *, cityInfo, charCompare> graph;
```

Once more we have used a typedef, since all the functionality we desire is provided already by the map data abstraction. We can as follows create an instance of graph and initialize it with the information described in the graph in Figure 15.1:

```
graph cityMap;

cityMap["Pendleton"]["Phoenix"] = 4;
cityMap["Pendleton"]["Pueblo"] = 8;
cityMap["Pensacola"]["Phoenix"] = 5;
cityMap["Peoria"]["Pittsburgh"] = 5;
cityMap["Peoria"]["Pueblo"] = 3;
cityMap["Phoenix"]["Peoria"] = 4;
cityMap["Phoenix"]["Pittsburgh"] = 10;
cityMap["Phoenix"]["Pueblo"] = 3;
cityMap["Pierre"]["Pendleton"] = 2;
cityMap["Pittsburgh"]["Pensacola"] = 4;
cityMap["Princeton"]["Pittsburgh"] = 2;
cityMap["Pueblo"]["Pierre"] = 3;
```

The first subscript indexes the graph, and returns a cityInfo, creating a new cityInfo if no such value exists already. The second subscript is then applied to the cityInfo, creating a new position for an unsigned integer value. The assignment then changes the association in the cityInfo map. The type graph is, in effect, a two-dimensional sparse array, indexed by strings and holding integer values.

### 15.1.1 Shortest Path Algorithm

We now turn our attention to the problem of finding the shortest path to each reachable city, starting from a given initial location. The algorithm we use is a well known technique, named *Dijkstra's Algorithm* in honor of the computer scientist credited with its discovery.

The idea of Dijkstra's algorithm is to start with a city of origin, and make a list of the cities that can be reached in one step. Order this list by cost, with the least costly city listed first.

Remove the first element from this list. This cannot help but be the least costly way to reach this first city, since any other path to the city would have to be along a path that begins in another reachable city, and we know that all other reachable cities are more costly.

Now comes the key insight. Determine the cities that are reachable from this first destination, and add the costs of travel for each to the cost of making the first leg. Using these combined cost figures, add these new destinations to our list of reachable locations, one more keeping the list ordered by the total cost.

To complete the algorithm, we need only put a loop around this operation, and note that we need not consider a city when it reaches the top of the list if we have already discovered a less costly way to reach the city.

### 15.1.2 Developing the Data Structures

The final result we desire is a list of cities, and the cost to travel to each. We can use the cityInfo data type defined earlier to hold this information. Let us use the name travelCosts for this data structure.

The list discussed in the informal description consists of entries that hold two values, a name and a cost. There is no ready made data type for this structure, so we are forced to define a new class. The constructor for the class will take a city name and a cost. Because some of the data structures in the STL require elements to have a default constructor, we provide one, although it will never be used in our algorithm. Because we want to be able to compare two such values, we override the comparison operator:

```
class Destination {
public:
    Destination () : distance(0) { }
    Destination (const char * dt, unsigned int ds)
        : distance(ds), destination(dt)  { }

    bool operator < (const Destination & right) const
        { return distance < right.distance; }

    unsigned int distance;
    const char * destination;
```

```
};
```

We have here overloaded the comparison operator as a member function. In the rational number case study described in Chapter 14 we illustrated overloading the comparison operator as an ordinary (that is, non-member) function.

Remember that we wanted to keep the list ordered by cost, least to first. This action will be performed for us automatically if we use a *priority queue*. The priority_queue data type in the STL requires two template arguments, the first indicating an underlying container to use for holding the actual values, and the second indicating the operation used in comparing values. We can use a vector for the first, and a library provided function object named lesser for the second. (lesser is a function object that invokes the comparison operator for our data type, and eliminates the need to define a special function object). The queue is initialized with a single entry, corresponding to a "trip" with no cost to the initial city.

DEFINE *A priority queue maintains elements in order, providing fast access to the topmost value*

```
priority_queue< vector<Destination>, lesser<Destination> > que;
    // put starting city in queue
que.push (Destination(startingCity, 0));
```

At each step of the algorithm we pull an entry from the priority queue, and ask whether or not we have yet visited this city. There is no direct way to determine if a map has an entry under a given key, however the information can be indirectly inferred. We do this by counting the number of entries in the cost map that have the new city as a key. If this count is zero, then we have not yet visited the city.

```
    // remove top entry from queue
char * newCity = que.top().destination;
int cost = que.top().distance;
que.pop();
if (travelCosts.count(newCity) == 0) {
    ... // have not seen it yet
}
```

If we have not been to the city, an entry is made in the travelCosts map:

```
travelCosts[newCity] = cost;
```

Next we want to add to the priority queue the cities that are reachable from the new city. To do that, we create iterators that cycle over the city information map associated with the new city. Recall that iterators for a map data type yield values of type Pair. The key field in such a value is obtained as the field named first, while the value portion is found in a field named second. Each step of the iteration, we add the cost to date to the new cost, and create a new destination entry:

```
    cityInfo::iterator start = cityMap[newCity].begin();
    cityInfo::iterator stop = cityMap[newCity].end();
    for (; start != stop; ++start) {
        const char * destCity = (*start).first;
            // make the new routine
        unsigned int destDistance = (*start).second;
        que.push(Destination(destCity, cost + destDistance));
    }
```

We can put everything together in the algorithm shown in Figure 15.2. Note how, in this one algorithm, we have made use of the following STL collections: map, vector, priority_queue, as well as the function object lesser.

To complete the program, we need a main procedure. The following double nested loop will print the cost of travel from each city to every other reachable city.

```
int main()
{

    graph cityMap;
    ... // initialization of the map

    graph::iterator start = cityMap.begin();
    graph::iterator stop = cityMap.end();
    for ( ; start != stop; ++start) {
        const char * city = (*start).first;
        cout << "\nStarting from " << city << "\n";
        cityInfo costs;
        dijkstra(cityMap, city, costs);
        cityInfo::iterator cstart = costs.begin();
        cityInfo::iterator cstop = costs.end();
        for ( ; cstart != cstop; ++cstart) {
            cout << "to " << (*cstart).first <<
                " costs " << (*cstart).second << '\n';
        }
    }
    return 0;
}
```

```
void dijkstra(graph cityMap, const char * start, cityInfo & travelCosts)
    // dijkstra's single source shortest path algorithm
{
        // keep a priority queue of distances to cities
    priority_queue < vector<Destination>, lesser<Destination> > que;
    que.push (Destination(start, 0));

        // while queue not empty
    while (! que.empty() ) {
            // remove top entry from queue
        const char * newCity = que.top().destination;
        int cost = que.top().distance;
        que.pop();
            // if so far unvisited,
        if (travelCosts.count(newCity) == 0) {
                // visit it now
            travelCosts[newCity] = cost;
                // add reachable cities to list
            cityInfo::iterator start = cityMap[newCity].begin();
            cityInfo::iterator stop = cityMap[newCity].end();
            for (; start != stop; ++start) {
                const char * destCity = (*start).first;
                unsigned int destDistance = (*start).second;
                que.push(Destination(destCity, cost + destDistance));
            }
        }
    }
}
```

Figure 15.2: Dijkstras Shortest Path Algorithm

## 15.2 A Concordance

Our second example program to illustrate the use of the STL collection data abstractions will be a concordance. A concordance is an alphabetical listing of words in a text, that indicates the line numbers on which each word occurs. The data values will be maintained in the concordance by a map, indexed by strings (the words) and holding sets of integers (the line numbers). A set is employed for the value stored under each key because the same word will often appear on multiple different lines; indeed, discovering such connections is one of the primary purposes of a concordance.

```
class concordance {
    typedef set<int, less<int> > lineList;
    typedef map<string, lineList, less<string> > wordDictType;
public:
    void readText (istream &);
    void printConcordance (ostream &);

protected:
    wordDictType wordMap;
};
```

Note that the class definition does not include a constructor function. In such situations a default constructor will be automatically created, and this will in turn invoke the default constructor for the wordMap data field. The default constructor for a map creates a collection with no entries.

The creation of the concordance is divided into two steps: first the program generates the concordance (by reading lines from an input stream), and then the program prints the result on the output stream. This is reflected in the two member functions readText() and printConcordance(). The first of these, readText(), is written as follows:

```
void concordance::readText (istream & in)
    // read all words from input stream, entering into concordance
{
    string line;
    for (int i = 1; getline(in, line); i++) {
            // translate into lower case, split into words
        allLower(line);
        list<string> words;
        split(line, " ,.;:", words);
            // enter each word on line into concordance
        list<string>::iterator wptr;
```

```
        for (wptr = words.begin(); wptr != words.end(); ++wptr)
            wordMap[*wptr].insert(i);
    }
}
```

Lines are read from the input stream one by one. The text of the line is first converted into lower case, then the line is split into words, using the function split() described in Chapter 8. Each word is then entered into the concordance. Subscripting the map creates an entry for the line list, if one does not already exist. Using the insert method for sets, the word is then entered into the container.

The final step is to print the concordance. This is performed in the following fashion:

```
void concordance::printConcordance (ostream & out)
    // print concordance on the given output stream
{
    string lastword = "";
    wordDictType::iterator pairPtr;
    wordDictType::iterator stop = wordMap.end();
    for (pairPtr = wordMap.begin(); pairPtr != stop; ++pairPtr) {
        out << (*pairPtr).first << " ";
        lineList & lines = (*pairPtr).second;
        lineList::iterator wstart = lines.begin();
        lineList::iterator wstop = lines.end();
        for ( ; wstart != wstop; ++wstart)
            out << *wstart << " ";
        cout << endl;
    }
}
```

An iterator loop is used to cycle over the elements being maintained by the word list. Each new word generates a new line of output–thereafter line numbers appear separated by spaces. For each word, a nested iterator loop cycles over the line numbers. If, for example, the input was the text:

It was the best of times,
it was the worst of times.

The output, from best to worst, would be:

best: 1
it: 1 2
of: 1 2
the: 1 2

times: 1 2
was: 1 2
worst: 2

## Test Your Understanding

1. Explain the purpose of the function object required as the third argument in the `map` data type.

2. How is the `priority_queue` data abstraction differ from the `queue` data type?