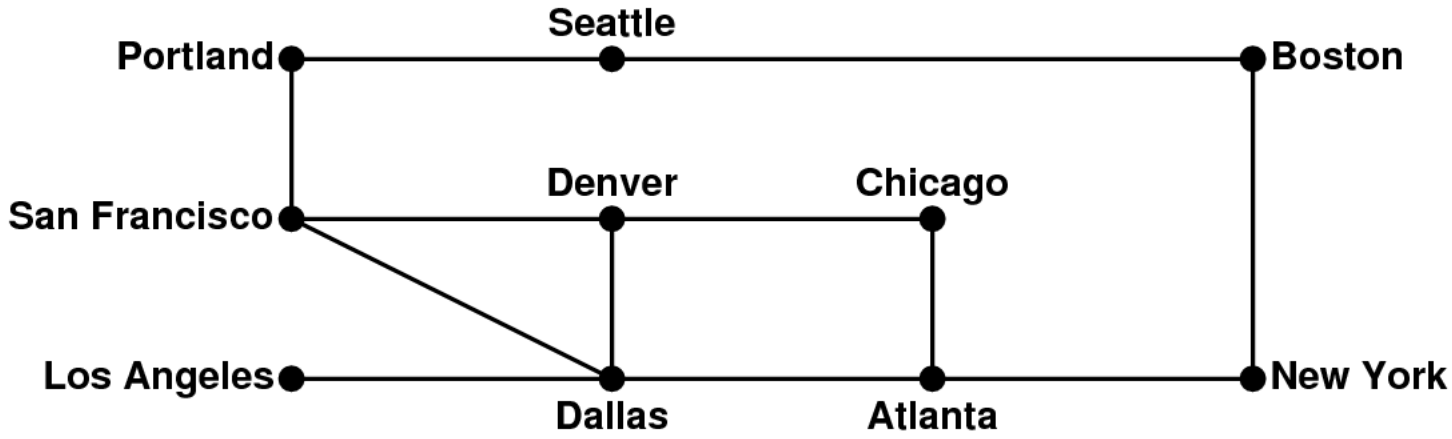
Graph Using Simple Graph and Using Graph Classes

Song Qi
221019037@link.cuhk.edu.cn

1. Graph Representation

- **The structure of a graph: nodes, arcs, and so on.**



- **How to represent a graph ?**
  - nodes: a set, vector, array.
  - arcs: map, structure types, classes designed.

2. Low-Level Graph Representation

- **SimpleGraph contains two sets and a map**

graphtypes.h

```
/*
 * Type: SimpleGraph
 * -----------------
 * This type represents a graph and consists of a set of nodes, a set of
 * arcs, and a map that creates an association between names and nodes.
 */

struct SimpleGraph {
   Set<Node *> nodes;
   Set<Arc *> arcs;
   Map<std::string,Node *> nodeMap;
};
```

- A **`map`** is included to translate node name into the corresponding node structure.

2. Low-Level Graph Representation

graphtypes.h

```
/* Type: SimpleGraph */

struct SimpleGraph {
   Set<Node *> nodes;
   Set<Arc *> arcs;
   Map<std::string,Node *> nodeMap;
};
```

- A structure type called **`Node`** that contains the **name** of the node and a set that indicates which **arcs** extend from the node to other nodes in the graph.

graphtypes.h

```
/*
 * Type: Node
 * ----------
 * This type represents an individual node and consists of the
 * name of the node and the set of arcs from this node.
 */

struct Node {
   std::string name;
   Set<Arc *> arcs;
};
```

2. Low-Level Graph Representation

graphtypes.h

```
/* Type: SimpleGraph */

struct SimpleGraph {
   Set<Node *> nodes;
   Set<Arc *> arcs;
   Map<std::string,Node *> nodeMap;
};
```

- A structure type called **`Arc`** specifying the **endpoints** of the arc, along with a numeric value representing the **cost**.

graphtypes.h

```
/*
 * Type: Arc
 * ---------
 * This type represents an individual arc and consists of pointers
 * to the endpoints, along with the cost of traversing the arc.
 */

struct Arc {
   Node *start;
   Node *finish;
   double cost;
};
```

2. Low-Level Graph Representation

- A example using the `graphtypes.h` interface

AirlineGraph.cpp

```
/* Function prototypes */

void printAdjacencyLists(SimpleGraph & g);
void initAirlineGraph(SimpleGraph & airline);
void addFlight(SimpleGraph & airline, string c1, string c2, int miles);
void addNode(SimpleGraph & g, string name);
void addArc(SimpleGraph & g, Node *n1, Node *n2, double cost);

/* Main program */

int main() {
   SimpleGraph airline;
   initAirlineGraph(airline);
   printAdjacencyLists(airline);
   return 0;
}
```

```
struct SimpleGraph {
   Set<Node *> nodes;
   Set<Arc *> arcs;
   Map<std::string,Node *> nodeMap;
};
```

2. Low-Level Graph Representation

- A example using the `graphtypes.h` interface

AirlineGraph.cpp

```
/* Function: addNode
 * Usage: addNode(g, name);
 * ------------------------
 * Adds a new node with the specified name to the graph.
 */
void addNode(SimpleGraph & g, string name) {
   Node *node = new Node;
   node->name = name;
   g.nodes.add(node);
   g.nodeMap[name] = node;
}

/* Function: addArc
 * Usage: addArc(g, n1, n2, cost);
 * -------------------------------
 * Adds a directed arc to the graph connecting n1 to n2.
 */
void addArc(SimpleGraph & g, Node *n1, Node *n2, double cost) {
   Arc *arc = new Arc;
   arc->start = n1;
   arc->finish = n2;
   arc->cost = cost;
   g.arcs.add(arc);
   n1->arcs.add(arc);
}
```

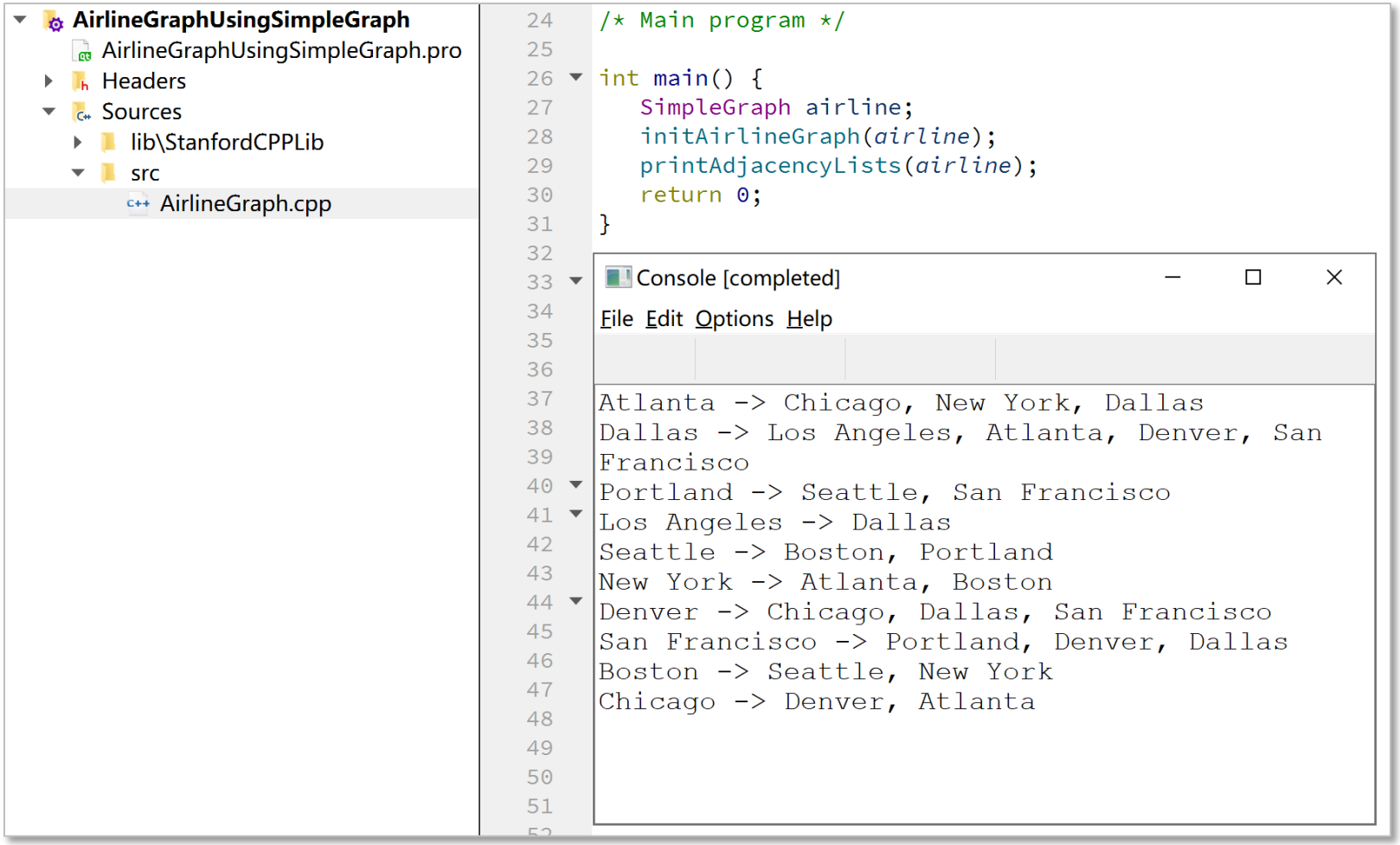
```
struct Node {
   std::string name;
   Set<Arc *> arcs;
};
```

```
struct Arc {
   Node *start;
   Node *finish;
   double cost;
};
```

```
struct SimpleGraph {
   Set<Node *> nodes;
   Set<Arc *> arcs;
   Map<std::string,Node *> nodeMap;
};
```

2. Low-Level Graph Representation

- A example using the `graphtypes.h` interface



3. Graph Class

- SimpleGraph uses low-level structure to represent a graph and **takes no advantage of (Objected Oriented Programming) OOP features of C++.**

- The Graph class is implemented as a **parameterized class (a template)** like conventional collection class.
  - With chosen types for nodes and arcs, a specific graph class will be generated and used.

3. Graph Class

graph.h

```
template <typename NodeType,typename ArcType>
class Graph {
...

private:

/* Instance variables */

   Set<NodeType *> nodes;                /* The set of nodes in the graph */
   Set<ArcType *> arcs;                  /* The set of arcs in the graph  */
   Map<std::string,NodeType *> nodeMap;  /* A map from names and nodes    */

/* Private methods */

   void deepCopy(const Graph & src);
   NodeType *getExistingNode(std::string name) const;

};
```

graphtypes.h

```
/* Type: SimpleGraph */

struct SimpleGraph {
   Set<Node *> nodes;
   Set<Arc *> arcs;
   Map<std::string,Node *> nodeMap;
};
```

graphtypes.h

```
struct Node {
   std::string name;
   Set<Arc *> arcs;
};
```

graphtypes.h

```
struct Arc {
   Node *start;
   Node *finish;
   double cost;
};
```

3. Graph Class

- **Graph Class**

graph.h

```
template <typename NodeType,typename ArcType>
class Graph {
...
private:
   Set<NodeType *> nodes;
   Set<ArcType *> arcs;
   Map<std::string,NodeType *> nodeMap;
...
};
```

- **Simple Graph**

graphtypes.h

```
/* Type: SimpleGraph */

struct SimpleGraph {
   Set<Node *> nodes;
   Set<Arc *> arcs;
   Map<std::string,Node *> nodeMap;
};
```

3. Graph Class

➢ The type the client chooses to represent a **node** must contain:

① A string field called **name** that specifies the name of the node;

② A field called **arcs** that specifies set of arcs that begin at this node.

AirlineGraph.cpp

```
/* Class: City
 * -----------
 * This class defines the node type.
 */

class City {
public:
   string getName() {
      return name;
   }
   void setCode(string code) {
       airportCode = code;
   }

private:
   string name;
   Set<Flight *> arcs;
   string airportCode;
   friend class Graph<City,Flight>;
};
```

- **Simple Graph**

graphtypes.h

```
struct Node {
   std::string name;
   Set<Arc *> arcs;
};
```

3. Graph Class

- **Graph Class**

graph.h

```
template <typename NodeType,typename ArcType>
class Graph {
...
private:
   Set<NodeType *> nodes;
   Set<ArcType *> arcs;
   Map<std::string,NodeType *> nodeMap;
...
};
```

- **Simple Graph**

graphtypes.h

```
/* Type: SimpleGraph */

struct SimpleGraph {
   Set<Node *> nodes;
   Set<Arc *> arcs;
   Map<std::string,Node *> nodeMap;
};
```

- The type the client chooses to represent an **arc** must contain:
  - Fields called **start** and **finish** that indicate the endpoints of the arc.

AirlineGraph.cpp

```
/* Class: Flight
 * -------------
 * This class defines the arc type.
 */

class Flight {
public:
   City *getStart() {
      return start;
   }

   City *getFinish() {
      return finish;
   }

   int getDistance() {
      return distance;
   }

   void setDistance(int miles) {
      distance = miles;
   }

private:
   City *start;
   City *finish;
   int distance;
   friend class Graph<City,Flight>;
};
```

- **Simple Graph**

graphtypes.h

```
struct Arc {
   Node *start;
   Node *finish;
   double cost;
};
```

3. Graph Class

- **Graph Class**

graph.h

```
template <typename NodeType,typename ArcType>
class Graph {
...
private:
   Set<NodeType *> nodes;
   Set<ArcType *> arcs;
   Map<std::string,NodeType *> nodeMap;
...
};
```

AirlineGraph.cpp

```
/* Main program */

int main() {
   Graph<City,Flight> airline;
   initAirlineGraph(airline);
   printAdjacencyLists(airline);
   return 0;
}
```

- **Simple Graph**

graphtypes.h

```
/* Type: SimpleGraph */

struct SimpleGraph {
   Set<Node *> nodes;
   Set<Arc *> arcs;
   Map<std::string,Node *> nodeMap;
};
```

AirlineGraph.cpp

```
/* Type: SimpleGraph */

int main() {
   SimpleGraph airline;
   initAirlineGraph(airline);
   printAdjacencyLists(airline);
   return 0;
}
```

3. Graph Class

- The public functions

graph.h

```
template <typename NodeType,typename ArcType>
class Graph {

public:
/* Constructor: Graph */
   Graph();

/* Destructor: ~Graph */
   ~Graph();

/* Method: size */
   int size() const;

/* Method: isEmpty */
   bool isEmpty() const;

/* Method: clear */
   void clear();

/* Method: addNode */
   NodeType *addNode(std::string name);
   NodeType *addNode(NodeType *node);
...
};
```

3. Graph Class

- The public methods in the interface.

Constructor

`Graph<`*nodetype*`,` *arctype*`>()`	Creates an empty graph with no nodes and no arcs.

Methods

`size()`	Returns the number of nodes in the graph.
`isEmpty()`	Returns `true` if the graph contains no nodes.
`clear()`	Removes all the nodes and arcs from the graph.
`addNode(`*name*`)` `addNode(`*node*`)`	Adds the node to the graph. The first form constructs a new node from the name; the second adds a node constructed by the client.
`removeNode(`*name*`)` `removeNode(`*node*`)`	Removes a node from the graph, along with all arcs involving that node.
`getNode(`*name*`)`	Returns the node associated with *name*. If no node exists with the specified name, `getNode` returns `NULL`.

`addArc(`s_1`,` s_2`)`
`addArc(`n_1`,` n_2`)`
`addArc(`*arc*`)`
	Adds an arc to the graph connecting the two nodes. The first two forms adds an arc connecting the specified nodes; the third form adds an arc constructed by the client.

`removeArc(`s_1`,` s_2`)`
`removeArc(`n_1`,` n_2`)`
`removeArc(`*arc*`)`
	Removes any arcs connecting the specified nodes.

`isConnected(`s_1`,` s_2`)` `isConnected(`n_1`,` n_2`)`	Returns `true` if there is an arc connecting the two nodes.
`getNodeSet()`	Returns the set of all nodes in a graph.
`getArcSet()`	Returns the set of all arcs in a graph.
`getArcSet(`*name*`)` `getArcSet(`*node*`)`	Returns the set of all arcs leaving the specified node.
`getNeighbors(`*name*`)` `getNeighbors(`*node*`)`	Returns the set of all nodes that are neighbors of the current node, in the sense that there is an arc from the specified node to the neighbor.

3. Graph Class

- **addNode :** add the node to the set of nodes for the graph and to the map from names to nodes.

➢ **Simple Graph** AirlineGraph.cpp

```
void addNode(SimpleGraph & g, string name) {
   Node *node = new Node;
   node->name = name;
   g.nodes.add(node);
   g.nodeMap[name] = node;
}
```

➢ **Graph Class** graph.h

```
template <typename NodeType,typename ArcType>
NodeType *Graph<NodeType,ArcType>::addNode(std::string name) {
   if (nodeMap.containsKey(name)) {
      error("addNode: Node " + name + " already exists");
   }
   NodeType *node = new NodeType();
   node->name = name;
   return addNode(node);
}

template <typename NodeType,typename ArcType>
NodeType *Graph<NodeType,ArcType>::addNode(NodeType *node) {
   nodes.add(node);
   nodeMap[node->name] = node;
   return node;
}
```

- **addArc :** Adds a directed arc to the graph connecting nodes.

➢ **Simple Graph** AirlineGraph.cpp

```
void addArc(SimpleGraph & g, Node *n1, Node *n2,
double cost) {
    Arc *arc = new Arc;
    arc->start = n1;
    arc->finish = n2;
    arc->cost = cost;
    g.arcs.add(arc);
    n1->arcs.add(arc);
}
```

➢ **Graph Class** graph.h

```
template <typename NodeType,typename ArcType>
ArcType *Graph<NodeType,ArcType>::addArc(std::string s1, std::string s2) {
    return addArc(getExistingNode(s1), getExistingNode(s2));
}

template <typename NodeType,typename ArcType>
ArcType *Graph<NodeType,ArcType>::addArc(NodeType *n1, NodeType *n2) {
    ArcType *arc = new ArcType();
    arc->start = n1;
    arc->finish = n2;
    return addArc(arc);
}

template <typename NodeType,typename ArcType>
ArcType *Graph<NodeType,ArcType>::addArc(ArcType *arc) {
    arc->start->arcs.add(arc);
    arcs.add(arc);
    return arc;
}
```

- **addArc :** Adds a directed arc to the graph connecting nodes.

➢ **Simple Graph** AirlineGraph.cpp

```
void addArc(SimpleGraph & g, Node *n1, Node *n2,
double cost) {
    Arc *arc = new Arc;
    arc->start = n1;
    arc->finish = n2;
    arc->cost = cost;
    g.arcs.add(arc);
    n1->arcs.add(arc);
}
```

➢ **Graph Class** AirlineGraph.cpp

```
void addFlight(Graph<City,Flight> & airline, string c1, string c2, int miles) {
    airline.addArc(c1, c2)->setDistance(miles);
    airline.addArc(c2, c1)->setDistance(miles);
}
```

AirlineGraph.cpp

```
class Flight {
public:
    ...
    void setDistance(int miles) {
        distance = miles;
    }

private:
    ...
};
```

End