// Map.h

#ifndef MAP\_INCLUDED

#define MAP\_INCLUDED

// Later in the course, we'll see that templates provide a much nicer

// way of enabling us to have Maps of different types. For now, we'll

// use typedefs.

#include <string>

**typedef std::string KeyType;**

**typedef double ValueType;**

const int DEFAULT\_MAX\_ITEMS = 200;

class Map

{

public:

Map(); // Create an empty map (i.e., one with no key/value pairs)

bool empty() **const**; // Return true if the map is empty, otherwise false.

int size() **const**; // Return the number of key/value pairs in the map.

bool insert(const **KeyType**& key, const **ValueType**& value);

// If key is not equal to any key currently in the map, and if the

// key/value pair can be added to the map, then do so and return true.

// Otherwise, make no change to the map and return false (indicating

// that either the key is already in the map, or the map has a fixed

// capacity and is full.

bool update(const **KeyType**& key, const **ValueType**& value);

// If key is equal to a key currently in the map, then make that key no

// longer map to the value it currently maps to, but instead map to

// the value of the second parameter; return true in this case.

// Otherwise, make no change to the map and return false.

bool insertOrUpdate(const **KeyType**& key, const **ValueType**& value);

// If key is equal to a key currently in the map, then make that key no

// longer map to the value it currently maps to, but instead map to

// the value of the second parameter; return true in this case.

// If key is not equal to any key currently in the map, and if the

// key/value pair can be added to the map, then do so and return true.

// Otherwise, make no change to the map and return false (indicating

// that the key is not already in the map and the map has a fixed

// capacity and is full.

bool erase(const **KeyType**& key);

// If key is equal to a key currently in the map, remove the key/value

// pair with that key from the map and return true. Otherwise, make

// no change to the map and return false.

bool contains(const **KeyType**& key) **const**;

// Return true if key is equal to a key currently in the map, otherwise

// false.

bool get(const **KeyType**& key, **ValueType**& value) **const**;

// If key is equal to a key currently in the map, set value to the

// value in the map that that key maps to and return true. Otherwise,

// make no change to the value parameter of this function and return

// false.

bool get(int i, **KeyType**& key, **ValueType**& value) **const**;

// If 0 <= i < size(), copy into the key and value parameters the

// key and value of one of the key/value pairs in the map and return

// true. Otherwise, leave the key and value parameters unchanged and

// return false.

void swap(Map& other);

// Exchange the contents of this map with the other one.

private:

// Since this structure is used only by the implementation of the

// Map class, we'll make it private to Map. Alternatively,

// we could have declared it outside the Map class, but then

// clients could use it, and there's no need for them to.

struct Pair

{

KeyType m\_key;

ValueType m\_value;

};

Pair m\_data[DEFAULT\_MAX\_ITEMS]; // the pairs in the map

int m\_size; // number of pairs in the map

// At any time, the elements of m\_data indexed from 0 to m\_size-1

// are in use.

int find(const KeyType& key) const;

// Return index of the pair in m\_data whose m\_key == key if there is

// one, else -1

bool doInsertOrUpdate(const KeyType& key, const ValueType& value,

bool mayInsert, bool mayUpdate);

// If the key is not present in the map and if mayInsert is true, insert

// the pair if there is room. If the key is present and mayUpdate is

// true, update the pair with the given key.

};

// Inline implementations

inline

int Map::size() const

{

return m\_size;

}

inline

bool Map::empty() const

{

return size() == 0;

}

inline

bool Map::contains(const KeyType& key) const

{

return find(key) != -1;

}

inline

bool Map::insert(const KeyType& key, const ValueType& value)

{

return doInsertOrUpdate(key, value, true /\* insert \*/, false /\* no update \*/);

}

inline

bool Map::update(const KeyType& key, const ValueType& value)

{

return doInsertOrUpdate(key, value, false /\* no insert \*/, true /\* update \*/);

}

inline

bool Map::insertOrUpdate(const KeyType& key, const ValueType& value)

{

return doInsertOrUpdate(key, value, true /\* insert \*/, true /\* update \*/);

}

#endif // MAP\_INCLUDED

===================================================================

// Map.cpp

#include "Map.h"

Map::Map()

: m\_size(0)

{}

bool Map::erase(const KeyType& key)

{

int pos = find(key);

if (pos == -1) // not found

return false;

// Move last array item to replace the one to be erased

m\_size--;

m\_data[pos] = m\_data[m\_size];

return true;

}

bool Map::get(const KeyType& key, ValueType& value) const

{

int pos = find(key);

if (pos == -1) // not found

return false;

value = m\_data[pos].m\_value;

return true;

}

bool Map::get(int i, KeyType& key, ValueType& value) const

{

if (i < 0 || i >= m\_size)

return false;

key = m\_data[i].m\_key;

value = m\_data[i].m\_value;

return true;

}

void Map::swap(Map& other)

{

// Swap elements. Since the only elements that matter are those up to

// m\_size and other.m\_size, only they have to be moved.

int minSize = (m\_size < other.m\_size ? m\_size : other.m\_size);

for (int k = 0; k < minSize; k++)

{

Pair tempPair = m\_data[k];

m\_data[k] = other.m\_data[k];

other.m\_data[k] = tempPair;

}

// If the sizes are different, assign the remaining elements from

// the longer one to the shorter.

if (m\_size > minSize)

for (int k = minSize; k < m\_size; k++)

other.m\_data[k] = m\_data[k];

else if (other.m\_size > minSize)

for (int k = minSize; k < other.m\_size; k++)

m\_data[k] = other.m\_data[k];

// Swap sizes

int t = m\_size;

m\_size = other.m\_size;

other.m\_size = t;

}

int Map::find(const KeyType& key) const

{

// Do a linear search through the array.

for (int pos = 0; pos < m\_size; pos++)

if (m\_data[pos].m\_key == key)

return pos;

return -1;

}

bool Map::doInsertOrUpdate(const KeyType& key, const ValueType& value,

bool mayInsert, bool mayUpdate)

{

int pos = find(key);

if (pos != -1) // found

{

if (mayUpdate)

m\_data[pos].m\_value = value;

return mayUpdate;

}

if (!mayInsert) // not found, and not allowed to insert

return false;

if (m\_size == DEFAULT\_MAX\_ITEMS) // no room to insert

return false;

m\_data[m\_size].m\_key = key;

m\_data[m\_size].m\_value = value;

m\_size++;

return true;

}

**Problem 4:**

// WeightMap.h

#ifndef WEIGHTMAP\_INCLUDED

#define WEIGHTMAP\_INCLUDED

#include "Map.h" // KeyType and ValueType are typedef'd to string and double.

#include <string>

class WeightMap

{

public:

WeightMap(); // Create an empty weight map.

bool enroll(std::string name, double startWeight);

// If a person with the given name is not currently in the map,

// there is room in the map, and the startWeight is not negative,

// add an entry for that person and weight and return true.

// Otherwise make no change to the map and return false.

double weight(std::string name) const;

// If a person with the given name is in the map, return that

// person's weight; otherwise, return -1.

bool adjustWeight(std::string name, double amt);

// If no person with the given name is in the map or if

// weight() + amt is negative, make no change to the map and return

// false. Otherwise, change the weight of the indicated person by

// the given amount and return true. For example, if amt is -8.2,

// the person loses 8.2 pounds; if it's 3.7, the person gains 3.7

// pounds.

int size() const; // Return the number of people in the WeightMap.

void print() const;

// Write to cout one line for every person in the map. Each line

// has the person's name, followed by one space, followed by that

// person's weight.

private:

Map m\_weightMap;

};

// Inline implementations

inline

bool WeightMap::enroll(std::string name, double startWeight)

{

return startWeight >= 0 && m\_weightMap.insert(name, startWeight);

}

inline

double WeightMap::weight(std::string name) const

{

double weight;

return m\_weightMap.get(name, weight) ? weight : -1;

}

inline

int WeightMap::size() const

{

return m\_weightMap.size();

}

#endif // WEIGHTMAP\_INCLUDED

===================================================================

// WeightMap.cpp

#include "Map.h"

#include "WeightMap.h"

#include <iostream>

using namespace std;

// Actually, we did not have to declare and implement the default

// constructor: If we declare no constructors whatsoever, the compiler

// writes a default constructor for us that would do nothing more than

// default construct the m\_weightMap data member.

// We do not have to declare a destructor, copy constructor, or assignment

// operator, because the compiler-generated ones do the right thing.

WeightMap::WeightMap()

{}

bool WeightMap::adjustWeight(string name, double amt)

{

double weight;

if (!m\_weightMap.get(name, weight))

return false;

weight += amt;

return weight >= 0 && m\_weightMap.update(name, weight);

}

void WeightMap::print() const

{

string name;

double weight;

for (int k = 0; k < m\_weightMap.size(); k++)

{

m\_weightMap.get(k, name, weight);

cout << name << ' ' << weight << endl;

}

}

**Problem 5:**

The few differences from the Problem 3 solution are indicated in boldface.

**// newMap.h**

**#ifndef NEWMAP\_INCLUDED**

**#define NEWMAP\_INCLUDED**

// Later in the course, we'll see that templates provide a much nicer

// way of enabling us to have Maps of different types. For now, we'll

// use typedefs.

#include <string>

typedef std::string KeyType;

typedef double ValueType;

const int DEFAULT\_MAX\_ITEMS = 200;

class Map

{

public:

**Map(int capacity = DEFAULT\_MAX\_ITEMS);**

**// Create an empty map with the given capacity.**

bool empty() const; // Return true if the map is empty, otherwise false.

int size() const; // Return the number of key/value pairs in the map.

bool insert(const KeyType& key, const ValueType& value);

// If key is not equal to any key currently in the map, and if the

// key/value pair can be added to the map, then do so and return true.

// Otherwise, make no change to the map and return false (indicating

// that either the key is already in the map, or the map has a fixed

// capacity and is full.

bool update(const KeyType& key, const ValueType& value);

// If key is equal to a key currently in the map, then make that key no

// longer map to the value it currently maps to, but instead map to

// the value of the second parameter; return true in this case.

// Otherwise, make no change to the map and return false.

bool insertOrUpdate(const KeyType& key, const ValueType& value);

// If key is equal to a key currently in the map, then make that key no

// longer map to the value it currently maps to, but instead map to

// the value of the second parameter; return true in this case.

// If key is not equal to any key currently in the map, and if the

// key/value pair can be added to the map, then do so and return true.

// Otherwise, make no change to the map and return false (indicating

// that the key is not already in the map and the map has a fixed

// capacity and is full.

bool erase(const KeyType& key);

// If key is equal to a key currently in the map, remove the key/value

// pair with that key from the map and return true. Otherwise, make

// no change to the map and return false.

bool contains(const KeyType& key) const;

// Return true if key is equal to a key currently in the map, otherwise

// false.

bool get(const KeyType& key, ValueType& value) const;

// If key is equal to a key currently in the map, set value to the

// value in the map that that key maps to and return true. Otherwise,

// make no change to the value parameter of this function and return

// false.

bool get(int i, KeyType& key, ValueType& value) const;

// If 0 <= i < size(), copy into the key and value parameters the

// key and value of one of the key/value pairs in the map and return

// true. Otherwise, leave the key and value parameters unchanged and

// return false.

void swap(Map& other);

// Exchange the contents of this map with the other one.

**// Housekeeping functions**

**~Map();**

**Map(const Map& other);**

**Map& operator=(const Map& rhs);**

private:

// Since this structure is used only by the implementation of the

// Map class, we'll make it private to Map. Alternatively,

// we could have declared it outside the Map class, but then

// clients could use it, and there's no need for them to.

struct Pair

{

KeyType m\_key;

ValueType m\_value;

};

**Pair\* m\_data; // dynamic array of the pairs in the map**

int m\_size; // number of pairs in the map

**int m\_capacity; // the maximum number of pairs**

// At any time, the elements of m\_data indexed from 0 to m\_size-1

// are in use.

int find(const KeyType& key) const;

// Return index of the pair in m\_data whose m\_key == key if there is

// one, else -1

bool doInsertOrUpdate(const KeyType& key, const ValueType& value,

bool mayInsert, bool mayUpdate);

// If the key is not present in the map and if mayInsert is true, insert

// the pair if there is room. If the key is present and mayUpdate is

// true, update the pair with the given key.

};

// Inline implementations

inline

int Map::size() const

{

return m\_size;

}

inline

bool Map::empty() const

{

return size() == 0;

}

inline

bool Map::contains(const KeyType& key) const

{

return find(key) != -1;

}

inline

bool Map::insert(const KeyType& key, const ValueType& value)

{

return doInsertOrUpdate(key, value, true /\* insert \*/, false /\* no update \*/);

}

inline

bool Map::update(const KeyType& key, const ValueType& value)

{

return doInsertOrUpdate(key, value, false /\* no insert \*/, true /\* update \*/);

}

inline

bool Map::insertOrUpdate(const KeyType& key, const ValueType& value)

{

return doInsertOrUpdate(key, value, true /\* insert \*/, true /\* update \*/);

}

**#endif // NEWMAP\_INCLUDED**

===================================================================

**// newMap.cpp**

**#include "newMap.h"**

**#include <iostream>**

**#include <cstdlib>**

**using namespace std;**

**Map::Map(int capacity)**

**: m\_size(0), m\_capacity(capacity)**

**{**

**if (capacity < 0)**

**{**

**cout << "A Map capacity must not be negative." << endl;**

**exit(1);**

**}**

**m\_data = new Pair[m\_capacity];**

**}**

**Map::Map(const Map& other)**

**: m\_size(other.m\_size), m\_capacity(other.m\_capacity)**

**{**

**m\_data = new Pair[m\_capacity];**

**for (int k = 0; k < m\_size; k++)**

**m\_data[k] = other.m\_data[k];**

**}**

**Map::~Map()**

**{**

**delete [] m\_data;**

**}**

**Map& Map::operator=(const Map& rhs)**

**{**

**if (this != &rhs)**

**{**

**Map temp(rhs);**

**swap(temp);**

**}**

**return \*this;**

**}**

bool Map::erase(const KeyType& key)

{

int pos = find(key);

if (pos == -1) // not found

return false;

// Move last array item to replace the one to be erased

m\_size--;

m\_data[pos] = m\_data[m\_size];

return true;

}

bool Map::get(const KeyType& key, ValueType& value) const

{

int pos = find(key);

if (pos == -1) // not found

return false;

value = m\_data[pos].m\_value;

return true;

}

bool Map::get(int i, KeyType& key, ValueType& value) const

{

if (i < 0 || i >= m\_size)

return false;

key = m\_data[i].m\_key;

value = m\_data[i].m\_value;

return true;

}

void Map::swap(Map& other)

{

**// Swap the m\_data pointers to dynamic arrays.**

**Pair\* tempData = m\_data;**

**m\_data = other.m\_data;**

**other.m\_data = tempData;**

// Swap sizes

int t = m\_size;

m\_size = other.m\_size;

other.m\_size = t;

**// Swap capacities.**

**t = m\_capacity;**

**m\_capacity = other.m\_capacity;**

**other.m\_capacity = t;**

}

int Map::find(const KeyType& key) const

{

// Do a linear search through the array.

for (int pos = 0; pos < m\_size; pos++)

if (m\_data[pos].m\_key == key)

return pos;

return -1;

}

bool Map::doInsertOrUpdate(const KeyType& key, const ValueType& value,

bool mayInsert, bool mayUpdate)

{

int pos = find(key);

if (pos != -1) // found

{

if (mayUpdate)

m\_data[pos].m\_value = value;

return mayUpdate;

}

if (!mayInsert) // not found, and not allowed to insert

return false;

if (m\_size == **m\_capacity**) // no room to insert

return false;

m\_data[m\_size].m\_key = key;

m\_data[m\_size].m\_value = value;

m\_size++;

return true;

}