```
 1: // Author: 500490778
 2: // Date: 8/23
 3:
 4: /*
 5: COccupancyMapHash is a class that inhertiances from the COccupancyMapBase class
 6: It creates a hash table to store observed/not observed locations. This was done in a format
 7: <key, hash>. The key is created by the MakeKey method which applies a function to the two
 8: ints for the location to create a unique double key.
 9: The class is able to add to the observed/not observed hash maps and check if a location
10: is occupied
11: */
12:
13:
14: #ifndef _OCCUPANCYMAPHASH_H
15: #define _OCCUPANCYMAPHASH_H
16:
17: #include "OccupancyMapBase.h"
18:
19: class COccupancyMapHash: public COccupancyMapBase
20: {
21:    public:
22:
23:     COccupancyMapHash();
24:     ~COccupancyMapHash(){};
25:
26:     // Return the name of the approach as a string, for display purposes
27:     std::string GetNameOfApproach();
28:
29:     // Add a location observed as occupied to the map
30:     void AddOccupiedLocation(std::pair<int,int> Location);
31:
32:     // Add a location observed as not occupided to the map
33:     void AddNotOccupiedLocation(std::pair<int,int> Location);
34:
35:     // Check if a location is occupied
36:     bool CheckIsOccupied( std::pair<int,int> Location);
37:
38:
39:
40:   private:
41:     std::string _Name; //name of mapping method
42:
43:     //hash table format, <key, hash>
44:     std::unordered_map<double, double> _ObservedMap;
45:     std::unordered_map<double, double> _NotObservedMap;
46:
47:     //key for unordered map, function that takes two ints and outputs a unique key for map
48:     double MakeKey(std::pair<int,int> Location);
49: };
50:
51:
52: #endif
```

```cpp
// Author: 500490778
// Date: 31/8/23

/*
Inhertiance class from COccupancyMapBase
*/


#include "OccupancyMapHash.h"

#include <iostream>
#include <string>
#include <utility>        // std::pair
#include <vector>

//initilise class, give it a name
COccupancyMapHash::COccupancyMapHash()
    :_Name("Hash table-based approach") {};


// Return the name of the approach as a string, for display purposes
std::string COccupancyMapHash::GetNameOfApproach() {
    return _Name;
};

//returns unique key from function with two ints
double COccupancyMapHash::MakeKey(std::pair<int,int> Location) {
    //ints between 0-2047, a/2048 + b creates unique key
    return ((double)Location.first/2048 + (double)Location.second);
};

// Add a location observed as occupied to the map
void COccupancyMapHash::AddOccupiedLocation(std::pair<int,int> Location) {
    //adds value into map, map[key] = hash
    _ObservedMap[MakeKey(Location)] = std::hash<double>()(MakeKey(Location));
};

// Add a location observed as not occupided to the map
void COccupancyMapHash::AddNotOccupiedLocation(std::pair<int,int> Location) {
    //adds value into map, map[key] = hash
    _NotObservedMap[MakeKey(Location)] = std::hash<double>()(MakeKey(Location));
};


// Check if a location is occupied
bool COccupancyMapHash::CheckIsOccupied( std::pair<int,int> Location ) {

    //output check
    bool check = true;

    //checks if key has a hash, if not return false
    if(_ObservedMap.find(MakeKey(Location)) == _ObservedMap.end()) {
        check = false;
    }

    return check;
};

```

```
    1: // OccupancyMap_Base.cpp
    2: //
    3: // Base class for an occupancy map
    4: //
    5: // Initial revision: Donald G Dansereau, 2022
    6:
    7: #include "OccupancyMapBase.h"
    8:
    9: #include <iostream>
   10: #include <string>
   11: #include <utility>        // std::pair
   12: #include <vector>
   13: #include <chrono>         // for measuring duration of function call
   14: #include <fstream>        // ifstream
   15:
   16: //--Load up a file and populate the map for testing-----------------------------------------------------
   17: void COccupancyMapBase::PopulateForEval( std::string ObservationsFName, std::string NotObservedFName )
   18: {
   19:   // Load the test points corresponding to observed occupied points in space
   20:   // Note these go both into the base class record of points for testing, and into the map being evaluated
   21:   {
   22:     std::cout << "Loading the file of observations... " << ObservationsFName << std::endl;
   23:     std::ifstream InFile( ObservationsFName );
   24:     std::pair<int,int> ReadVal;
   25:
   26:     while( InFile.good() )
   27:     {
   28:       InFile >> ReadVal.first;
   29:       InFile >> ReadVal.second;
   30:
   31:       if( InFile.eof() )
   32:       {
   33:         break;
   34:       }
   35:
   36:       mObservedPoints.push_back( ReadVal );  // add to the tester's store of observed points
   37:       AddOccupiedLocation( ReadVal );        // add to the derived class store of observed points
   38:       // std::cout << "Read " << ReadVal.first << " " << ReadVal.second << std::endl;
   39:     }
   40:     std::cout << "PopulateForEval loaded " << mObservedPoints.size() << " observed test points" << std::end
l;
   41:   }
   42:
   43:   // Load the test points corresponding to free space
   44:   // Note these go only into the base class record of points for testing, not the map being evaluated
   45:   {
   46:     std::cout << "Loading the file of test points that don't correspond to observations... " << NotObserved
FName << std::endl;
   47:     std::ifstream InFile( NotObservedFName );
   48:     std::pair<int,int> ReadVal;
   49:
   50:     while( InFile.good() )
   51:     {
   52:       InFile >> ReadVal.first;
   53:       InFile >> ReadVal.second;
   54:
   55:       if( InFile.eof() )
   56:       {
   57:         break;
   58:       }
   59:
   60:       mNotObservedPoints.push_back( ReadVal );  // add to the tester's store of unobserved points
   61:       // std::cout << "Read " << ReadVal.first << " " << ReadVal.second << std::endl;
   62:     }
   63:     std::cout << "PopulateForEval loaded " << mNotObservedPoints.size() << " unobserved test points" << std
::endl;
   64:   }
   65:
   66: }
   67:
   68: //--Evaluate the performance of the derived class-------------------------------------------------------
-
   69: void COccupancyMapBase::EvalPerformance( std::string ObservationsFName, std::string NotObservedFName )
   70: {
   71:   std::cout << "Evaluating " << GetNameOfApproach() << " using test points from "
   72:             << ObservationsFName << " and " << NotObservedFName
   73:             << std::endl;
   74:
   75:   // Populate the map
   76:   PopulateForEval( ObservationsFName, NotObservedFName );
   77:
   78:   // Evaluate
   79:   double MeanRuntime = FindTotRuntime();
```

```
   80:
   81:   // Report
   82:   std::cout << "Total time to test all points: " << MeanRuntime << " ms" << std::endl;
   83: };
   84:
   85:
   86: //--Helper function to validate and time-----------------------------------------------------------
   87: double COccupancyMapBase::FindTotRuntime()
   88: {
   89:   std::cout << "Measuring runtime..." << std::endl;
   90:   double Result = -1;
   91:
   92:   // Start a timer
   93:   auto t1 = std::chrono::high_resolution_clock::now();
   94:
   95:   // Test recall of existing occupied locations
   96:   for( auto TestPoint: mObservedPoints )
   97:   {
   98:     // std::cout << "Checking " << TestPoint.first << " " << TestPoint.second << std::endl;
   99:     bool TestResult = CheckIsOccupied( TestPoint );   // call the derived-class occupancy check
  100:     if( TestResult != true )                          // these should all return "true"
  101:     {
  102:       std::cout << "ERROR!" << std::endl;
  103:       std::cout << "  Observed point at " << TestPoint.first << " " << TestPoint.second << " claims to be u
noccupied" << std::endl;
  104:     }
  105:   }
  106:
  107:   // Test that unobserved points are also treated correctly
  108:   for( auto TestPoint: mNotObservedPoints )
  109:   {
  110:     // std::cout << "Checking " << TestPoint.first << " " << TestPoint.second << std::endl;
  111:     bool TestResult = CheckIsOccupied( TestPoint );   // call the derived-class occupancy check
  112:     if( TestResult != false )                         // these should all return "false"
  113:     {
  114:       std::cout << "ERROR!" << std::endl;
  115:       std::cout << "  Unobserved point at " << TestPoint.first << " " << TestPoint.second << " claims to be
 occupied" << std::endl;
  116:     }
  117:   }
  118:
  119:   // Stop the timer and compute time elapsed
  120:   auto t2 = std::chrono::high_resolution_clock::now();  // keep track of time
  121:   std::chrono::duration<double, std::milli> Duration = t2 - t1;
  122:
  123:   // Return timing result
  124:   Result = Duration.count();
  125:   return Result;
  126: }
  127:
```

```
    1: // OccupancyMap_Base.h
    2: //
    3: // Base class for an occupancy map
    4: //
    5: // This defines the interface for an occupancy map that can :
    6: // 1) record a set of observations as known occupied locations, and
    7: // 2) test if a given location has been observed as being occupied
    8: //
    9: // The base class knows how to evaluate the performance of a given occupancy map implementation by :
   10: // 1) loading a pair of test files, one defining a list of test points corresponding to observed occupied
   11: // locations, and one defining a set of test points corresponding to unoccupied space,
   12: // 2) populating the occupancy map with observed points by calling its AddOccupiedLocation function, and
   13: // 3) testing the occupancy map by checking that it returns correct values for the loaded test data
   14: //
   15: // The test also reports on how long it takes to run through all the test points, to allow a comparison
   16: // of speed.
   17: //
   18: // Initial revision: Donald G Dansereau, 2022
   19: // Final revision: Anonomous 2022
   20:
   21: #ifndef _OCCUPANCYMAPBASE_H
   22: #define _OCCUPANCYMAPBASE_H
   23:
   24: #include <string>
   25: #include <utility>        // std::pair
   26: #include <vector>
   27: #include <algorithm>      //std::find
   28: #include <unordered_map>  //obviously for hash map
   29:
   30: //-----------------------------------------------------------------------------------------------------------
-
   31: // Base class for an occupancy map.
   32: // An occupancy map accepts observed points, one at a time, to keep track of what parts of space are occupi
ed.
   33: // It offers a function to check if a given point in space is occupied.
   34: //
   35: // This implementation works on coordinates provided as std::pair<int,int>.
   36: // It is acceptable to assume all observed points will be integer, non-negative, and less than 2048
   37: //
   38: // The base class includes a built-in function for evaluating the correctness and speed of a particular der
ived
   39: // class implementation of the occupancy map. See EvalPerformance() for details.
   40:
   41: class COccupancyMapBase
   42: {
   43:   public:
   44:     //virtual ~COccupancyMapBase();
   45:
   46:     // Return the name of the approach as a string, for display purposes
   47:     virtual std::string GetNameOfApproach() = 0;
   48:
   49:     // Add a location observed as occupied to the map
   50:     virtual void AddOccupiedLocation(std::pair<int,int> Location) = 0;
   51:
   52:     // Add a location observed as not occupided to the map
   53:     virtual void AddNotOccupiedLocation(std::pair<int,int> Location) = 0;
   54:
   55:     // Check if a location is occupied
   56:     virtual bool CheckIsOccupied( std::pair<int,int> Location ) = 0;
   57:
   58:     // Check if a location is not occupied
   59:
   60:     //---Evaluation-----------------------------------------------------------------------------------------
-
   61:     // Evaluate the performance of an occupancy map.
   62:     // Loads a local file called ObservationsFName containing observations of occupied map locations.
   63:     // These are loaded into the map via the derived class AddOccupiedLocation function,
   64:     // then validated via the derived class CheckIsOccupied function.
   65:     // If any unexpected results are found, these are output to std::cout.
   66:     // The whole process is timed, and the timing results printed to std::cout.
   67:     void EvalPerformance( std::string ObservationsFName, std::string NotObservedFName );
   68:
   69:   protected:
   70:     //--Data-----------------------------------------------------------------------------------------------
-
   71:     std::vector<std::pair<int,int>> mObservedPoints;    // A set of test points that correspond to obstacle
s
   72:     std::vector<std::pair<int,int>> mNotObservedPoints; // A set of test points that correspond to free spa
ce
   73:
   74:     //--Helper Functions-----------------------------------------------------------------------------------
-
   75:
```

```
76:     // load a file of test points for testing
77:     void PopulateForEval( std::string ObservationsFName, std::string NotObservedFName );
78:
79:     // validate and time a derived-class map, returns runtime in ms
80:     double FindTotRuntime();
81: };
82:
83:
84:
85: #endif
86:
```

```cpp
//Author: 500490778
//Date: 8/22

/*
Inhertiance class from COccupancyMapBase
*/

#include "OccupancyMapVec.h"

#include <iostream>
#include <string>
#include <utility>        // std::pair
#include <vector>



//Initilises class
COccupancyMapVec::COccupancyMapVec()
  : _Name("Vector approach") {};

//returns the name of approach from user
std::string COccupancyMapVec::GetNameOfApproach() {
  return _Name;
};

// adds occupied location to vector occupancy map
void COccupancyMapVec::AddOccupiedLocation(std::pair<int,int> Location) {
  mObservedPoints.push_back(Location);
};

// adds not occupied location to vector map
void COccupancyMapVec::AddNotOccupiedLocation(std::pair<int,int> Location) {
  mNotObservedPoints.push_back(Location);
};

// checks vector map for if a location is present
bool COccupancyMapVec::CheckIsOccupied( std::pair<int,int> Location ) {

  bool IsOcc = false; //assume false

  //searches for location in vector map, returns true if location found
  if(std::find(mObservedPoints.begin(), mObservedPoints.end(), Location) != mObservedPoints.end()) {
    IsOcc = true;
  }

  return IsOcc;
};
```

```cpp
// Author: SID 500490778
// Date: 8/22

/*
This file runs a comparision between using a hash table vs a vector table for mapping an occupancy map
The time to test all points is compared by running through all test points
Required to include .h and .cpp files of includes below. Required occupancy and
not occupied observation text files in the form int, int on each line.
*/


#include "OccupancyMapBase.h"
#include "OccupancyMapVec.h"
#include "OccupancyMapHash.h"
#include <iostream>

//------------------------------------------------------------------------------
int main()
{

  //test for vector approach
  {
    COccupancyMapVec VecMap;
    VecMap.EvalPerformance( "ExampleObservations_Small.txt", "ExampleNotObserved_Small.txt" );
  }


  // test for Hashing approach
  {
    COccupancyMapHash HashMap;
    HashMap.EvalPerformance( "ExampleObservations_Small.txt", "ExampleNotObserved_Small.txt" );
  }

  return 0;
}

```

```
 1: // Author: 500490778
 2: // Date: 8/23
 3:
 4: /*
 5: COccupancyMapVec is a class that inhertiances from the COccupancyMapBase class
 6: It creates a vector of pairs to store observed/not observed locations in a ,int,int> format.
 7: The class is able to add to the observed/not observed vector maps and check if a location
 8: is occupied from the vector map
 9: */
10:
11: #ifndef _OCCUPANCYMAPVEC_H
12: #define _OCCUPANCYMAPVEC_H
13:
14: #include "OccupancyMapBase.h"
15:
16: class COccupancyMapVec: public COccupancyMapBase
17: {
18:   public:
19:
20:     COccupancyMapVec();
21:     ~COccupancyMapVec(){};
22:
23:      // Return the name of the approach as a string, for display purposes
24:     std::string GetNameOfApproach();
25:
26:     // Add a location observed as occupied to the map
27:     void AddOccupiedLocation(std::pair<int,int> Location);
28:
29:     // Add a location observed as not occupided to the map
30:     void AddNotOccupiedLocation(std::pair<int,int> Location);
31:
32:     // Check if a location is occupied
33:     bool CheckIsOccupied( std::pair<int,int> Location );
34:
35:   private:
36:     std::string _Name;
37: };
38:
39:
40: #endif
```