**Homework 3 Solution**

|  |  |
| --- | --- |
| [Problem 1](http://www.cs.ucla.edu/classes/spring12/cs32/Homeworks/3/solution.html#P1) | [Problem 3](http://www.cs.ucla.edu/classes/spring12/cs32/Homeworks/3/solution.html#P3) |
| [Problem 2](http://www.cs.ucla.edu/classes/spring12/cs32/Homeworks/3/solution.html#P2) | [Problem 4](http://www.cs.ucla.edu/classes/spring12/cs32/Homeworks/3/solution.html#P4) |

**Problem 1:**

Since all Landmarks have a name, and since the way you find out the name is going to be the same for all kinds of landmarks, the name should be a data member of Landmark, and the function to retrieve it need not be virtual.

The different kinds of landmarks may have different icons and different colors, so functions to report them should be virtual. Since most landmark types have yellow icons, it's convenient to have Landmark::color have an implementation (that returns "yellow") that derived classes may inherit if they wish. There is no reasonable default behavior for the icon retrieval function, so this should be pure virtual.

Observe how the constructors for the derived classes pass the name to the Landmark constructor. Examine Restaurant's constructor especially.

// ======================================= Landmark

class Landmark

{

public:

Landmark(string nm);

string name() const;

virtual string icon() const = 0;

virtual string color() const;

virtual ~Landmark() {}

private:

string m\_name;

};

Landmark::Landmark(string nm)

: m\_name(nm)

{}

string Landmark::name() const

{

return m\_name;

}

string Landmark::color() const

{

return "yellow";

}

// ======================================= Hotel

class Hotel : public Landmark

{

public:

Hotel(string nm);

virtual string icon() const;

virtual ~Hotel();

private:

};

Hotel::Hotel(string nm)

: Landmark(nm)

{}

string Hotel::icon() const

{

return "bed";

}

Hotel::~Hotel()

{

cout << "Destroying the hotel " << name() << "." << endl;

}

// ======================================= Restaurant

class Restaurant : public Landmark

{

public:

Restaurant(string nm, int cap);

virtual string icon() const;

virtual ~Restaurant();

private:

int m\_capacity;

};

Restaurant::Restaurant(string nm, int cap)

: Landmark(nm), m\_capacity(cap)

{}

string Restaurant::icon() const

{

if (m\_capacity < 40)

return "small knife/fork";

else

return "large knife/fork";

}

Restaurant::~Restaurant()

{

cout << "Destroying the restaurant " << name() << "." << endl;

}

// ======================================= Hospital

class Hospital : public Landmark

{

public:

Hospital(string nm);

virtual string icon() const;

virtual string color() const;

virtual ~Hospital();

};

Hospital::Hospital(string nm)

: Landmark(nm)

{}

string Hospital::icon() const

{

return "H";

}

string Hospital::color() const

{

return "blue";

}

Hospital::~Hospital()

{

cout << "Destroying the hospital " << name() << "." << endl;

}

**Problem 2:**

// Return true if any of the strings in the array is empty, false

// otherwise.

bool anyEmpty(const string a[], int n)

{

if (n <= 0)

return false;

if (a[0].empty())

return true;

return anyEmpty(a+1, n-1);

}

// Return the number of empty strings in the array.

int countEmpties(const string a[], int n)

{

if (n <= 0)

return 0;

int t = a[0].empty(); // 1 if true, 0 if false

return t + countEmpties(a+1, n-1);

}

// Return the subscript of the first empty string in the array.

// If no element is empty, return -1.

int firstEmpty(const string a[], int n)

{

if (n <= 0)

return -1;

if (a[0].empty())

return 0;

int k = firstEmpty(a+1, n-1);

if (k == -1)

return -1;

return 1 + k; // element k of "the rest of a" is element 1+k of a

}

// Return the subscript of the least string in the array (i.e.,

// the smallest subscript m such that there is no k for which

// a[k] < a[m]. If the array has no elements to examine, return -1.

int indexOfLeast(const string a[], int n)

{

if (n <= 0)

return -1;

if (n == 1)

return 0;

int k = 1 + indexOfLeast(a+1, n-1); // indexOfLeast can't return -1 here

return a[0] <= a[k] ? 0 : k;

}

// If all n2 elements of a2 appear in the n1 element array a1, in

// the same order (though not necessarily consecutively), then

// return true; otherwise (i.e., if the array a1 does not include

// a2 as a not-necessarily-contiguous subsequence), return false.

// (Of course, if a2 is empty (i.e., n2 is 0), return true.)

// For example, if a1 is the 7 element array

// "stan" "kyle" "cartman" "kenny" "kyle" "cartman" "butters"

// then the function should return true if a2 is

// "kyle" "kenny" "butters"

// or

// "kyle" "cartman" "cartman"

// and it should return false if a2 is

// "kyle" "butters" "kenny"

// or

// "stan" "kenny" "kenny"

bool includes(const string a1[], int n1, const string a2[], int n2)

{

if (n2 <= 0)

return true;

if (n1 < n2)

return false;

// If we get here, a1 and a2 are nonempty

if (a1[0] == a2[0])

return includes(a1+1, n1-1, a2+1, n2-1); // rest of a1, rest of a2

else

return includes(a1+1, n1-1, a2, n2); // rest of a1, all of a2

}

**Problem 3:**

bool pathExists(char maze[][10], int sr, int sc, int er, int ec)

{

if (sr == er && sc == ec)

return true;

maze[sr][sc] = '@'; // anything non-'.' will do

if (maze[sr-1][sc] == '.' && pathExists(maze, sr-1, sc, er, ec))

return true;

if (maze[sr+1][sc] == '.' && pathExists(maze, sr+1, sc, er, ec))

return true;

if (maze[sr][sc-1] == '.' && pathExists(maze, sr, sc-1, er, ec))

return true;

if (maze[sr][sc+1] == '.' && pathExists(maze, sr, sc+1, er, ec))

return true;

return false;

}

or

bool pathExists(char maze[][10], int sr, int sc, int er, int ec)

{

if (maze[sr][sc] != '.')

return false;

if (sr == er && sc == ec)

return true;

maze[sr][sc] = '@'; // anything non-'.' will do

if (pathExists(maze, sr-1, sc, er, ec))

return true;

if (pathExists(maze, sr+1, sc, er, ec))

return true;

if (pathExists(maze, sr, sc-1, er, ec))

return true;

if (pathExists(maze, sr, sc+1, er, ec))

return true;

return false;

}

**Problem 4:**

// Return the number of ways that all n2 elements of a2 appear

// in the n1 element array a1 in the same order (though not

// necessarily consecutively). The empty sequence appears in a

// sequence of length n1 in 1 way, even if n1 is 0.

// For example, if a1 is the 7 element array

// "stan" "kyle" "cartman" "kenny" "kyle" "cartman" "butters"

// then for this value of a2 the function must return

// "stan" "kenny" "cartman" 1

// "stan" "cartman" "butters" 2

// "kenny" "stan" "cartman" 0

// "kyle" "cartman" "butters" 3

int countIncludes(const string a1[], int n1, const string a2[], int n2)

{

if (n2 <= 0)

return 1;

if (n1 < n2)

return 0;

// If we get here, a1 and a2 are nonempty

int t = countIncludes(a1+1, n1-1, a2, n2); // rest of a1, all of a2

if (a1[0] == a2[0])

return t + countIncludes(a1+1, n1-1, a2+1, n2-1); // rest of a1, rest of a2

else

return t;

}

// Exchange two strings

void exchange(string& x, string& y)

{

string t = x;

x = y;

y = t;

}

// Rearrange the elements of the array so that all the elements

// whose value is < splitter come before all the other elements,

// and all the elements whose value is > splitter come after all

// the other elements. Upon return, firstNotLess is set to the index

// of the first element in the rearranged array that is >= splitter,

// or n if there is no such element, and firstGreater is set to

// the index of the first element that is > splitter, or n if

// there is no such element.

// In other words, upon return from the function, the array is a

// permutation of its original value such that

// \* for 0 <= i < firstNotLess, a[i] < splitter

// \* for firstNotLess <= i < firstGreater, a[i] == splitter

// \* for firstGreater <= i < n, a[i] > splitter

// All the elements < splitter end up in no particular order.

// All the elements > splitter end up in no particular order.

void split(string a[], int n, string splitter, int& firstNotLess, int& firstGreater)

{

if (n < 0)

n = 0;

// It will always be the case that just before evaluating the loop

// condition:

// firstNotLess <= firstUnknown and firstUnknown <= firstGreater

// Every element earlier than position firstNotLess is < splitter

// Every element from position firstNotLess to firstUnknown-1 is

// == splitter

// Every element from firstUnknown to firstGreater-1 is not

// known yet

// Every element at position firstGreater or later is > splitter

firstNotLess = 0;

firstGreater = n;

int firstUnknown = 0;

while (firstUnknown < firstGreater)

{

if (a[firstUnknown] > splitter)

{

firstGreater--;

exchange(a[firstUnknown], a[firstGreater]);

}

else

{

if (a[firstUnknown] < splitter)

{

exchange(a[firstNotLess], a[firstUnknown]);

firstNotLess++;

}

firstUnknown++;

}

}

}

// Rearrange the elements of the array so that

// a[0] <= a[1] <= a[2] <= ... <= a[n-2] <= a[n-1]

// If n <= 1, do nothing.

void order(string a[], int n)

{

if (n <= 1)

return;

// Split using a[0] as the splitter (any element would do).

int firstNotLess;

int firstGreater;

split(a, n, a[0], firstNotLess, firstGreater);

// sort the elements < splitter

order(a, firstNotLess);

// sort the elements > splitter

order(a+firstGreater, n-firstGreater);

}