Winter 2017 CS 32

**Homework 3 Solution**

|  |  |
| --- | --- |
| [Problem 1](http://web.cs.ucla.edu/classes/winter17/cs32/Homeworks/3/solution.html#P1) | [Problem 3](http://web.cs.ucla.edu/classes/winter17/cs32/Homeworks/3/solution.html#P3) |
| [Problem 2](http://web.cs.ucla.edu/classes/winter17/cs32/Homeworks/3/solution.html#P2) | [Problem 4](http://web.cs.ucla.edu/classes/winter17/cs32/Homeworks/3/solution.html#P4) |

**Problem 1:**

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

The different kinds of files may have different behavior for the open and redisplay functions, so these should be virtual. Since most files have the same behavior for redisplay, it's convenient to have File::redisplay have an implementation that derived classes may inherit if they wish. There is no reasonable default behavior for open, so this should be pure virtual.

Observe how the constructors for the derived classes pass the name to the File constructor. Examine Video's constructor especially. Also note that File, because it's designed as a base class, has a virtual destructor. Observe what happens if its destructor were not declared virtual.

// ======================================= File

class File

{

public:

File(string nm);

string name() const;

virtual void open() const = 0;

virtual void redisplay() const;

virtual ~File() {}

private:

string m\_name;

};

File::File(string nm)

: m\_name(nm)

{}

string File::name() const

{

return m\_name;

}

void File::redisplay() const

{

cout << "refresh the screen";

}

// ======================================= TextMsg

class TextMsg : public File

{

public:

TextMsg(string nm);

virtual void open() const;

virtual ~TextMsg();

};

TextMsg::TextMsg(string nm)

: File(nm)

{}

void TextMsg::open() const

{

cout << "open text message";

}

TextMsg::~TextMsg()

{

cout << "Destroying " << name() << ", a text message" << endl;

}

// ======================================= Video

class Video : public File

{

public:

Video(string nm, int playtime);

virtual void open() const;

virtual void redisplay() const;

virtual ~Video();

private:

int m\_playingTime;

};

Video::Video(string nm, int playtime)

: File(nm), m\_playingTime(playtime)

{}

void Video::open() const

{

cout << "play " << m\_playingTime << " second video";

}

void Video::redisplay() const

{

cout << "replay video";

}

Video::~Video()

{

cout << "Destroying " << name() << ", a video" << endl;

}

// ======================================= Picture

class Picture : public File

{

public:

Picture(string nm);

virtual void open() const;

virtual ~Picture();

};

Picture::Picture(string nm)

: File(nm)

{}

void Picture::open() const

{

cout << "show picture";

}

Picture::~Picture()

{

cout << "Destroying the picture " << name() << endl;

}

**Problem 2:**

// Return false if the somePredicate function returns false for at

// least one of the array elements; return true otherwise.

bool allTrue(const double a[], int n)

{

if (n <= 0)

return true;

if ( ! somePredicate(a[0]))

return false;

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

}

// Return the number of elements in the array for which the

// somePredicate function returns false.

int countFalse(const double a[], int n)

{

if (n <= 0)

return 0;

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

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

}

// Return the subscript of the first element in the array for which

// the somePredicate function returns false. If there is no such

// element, return -1.

int firstFalse(const double a[], int n)

{

if (n <= 0)

return -1;

if ( ! somePredicate(a[0]))

return 0;

int k = firstFalse(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 smallest double in the array (i.e.,

// the one whose value is <= the value of all elements). If more

// than one element has the same smallest value, return the smallest

// subscript of such an element. If the array has no elements to

// examine, return -1.

int indexOfMin(const double a[], int n)

{

if (n <= 0)

return -1;

if (n == 1)

return 0;

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

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

// Here's an alternative for the last two lines above:

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

// return a[k] <= a[n-1] ? k : n-1;

}

// 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

// 10 50 40 20 50 40 30

// then the function should return true if a2 is

// 50 20 30

// or

// 50 40 40

// and it should return false if a2 is

// 50 30 20

// or

// 10 20 20

bool includes(const double a1[], int n1, const double 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

// 10 50 40 20 50 40 30

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

// 10 20 40 1

// 10 40 30 2

// 20 10 40 0

// 50 40 30 3

int countIncludes(const double a1[], int n1, const double 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])

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

return t;

}

// Exchange two doubles

void exchange(double& x, double& y)

{

double t = x;

x = y;

y = t;

}

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

// whose value is > separator come before all the other elements,

// and all the elements whose value is < separator come after all

// the other elements. Upon return, firstNotGreater is set to the

// index of the first element in the rearranged array that is

// <= separator, or n if there is no such element, and firstLess is

// set to the index of the first element that is < separator, 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 < firstNotGreater, a[i] > separator

// \* for firstNotGreater <= i < firstLess, a[i] == separator

// \* for firstLess <= i < n, a[i] < separator

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

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

void separate(double a[], int n, double separator,

int& firstNotGreater, int& firstLess)

{

if (n < 0)

n = 0;

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

// condition:

// firstNotGreater <= firstUnknown and firstUnknown <= firstLess

// Every element earlier than position firstNotGreater is > separator

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

// == separator

// Every element from firstUnknown to firstLess-1 is not known yet

// Every element at position firstLess or later is < separator

firstNotGreater = 0;

firstLess = n;

int firstUnknown = 0;

while (firstUnknown < firstLess)

{

if (a[firstUnknown] < separator)

{

firstLess--;

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

}

else

{

if (a[firstUnknown] > separator)

{

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

firstNotGreater++;

}

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(double a[], int n)

{

if (n <= 1)

return;

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

int firstNotGreater;

int firstLess;

separate(a, n, a[0], firstNotGreater, firstLess);

// sort the elements > separator

order(a, firstNotGreater);

// sort the elements < separator

order(a+firstLess, n-firstLess);

}