```cpp
// power.cpp
// Create a power() function and test it

#include <fstream>
#include <iostream>
#include <iomanip>
#include "console.h"
#include "filelib.h"
#include "simpio.h"

using namespace std;

void testPower(int base, int exp, double expected);

double power(int base, int exponent) {
    if(exponent == 0) {
        // base case
        return 1;    // no trebble....
    } else if(exponent < 0) {
        // recursive case 1
        return 1.0 / power(base, -exponent);
    } else {
        // recursive case 2
        return base * power(base, exponent - 1);
    }
}

int main() {
    cout << "Recursive power" << endl;
    testPower(2, 5, 32);
    testPower(5, 5, 3125);
    testPower(0, 6, 0);
    testPower(-6, 3, -216);
    testPower(6, 0, 1);
    testPower(2, -3, 0.125);
    cout << "Done!" << endl;
    return 0;
}

void testPower(int base, int exponent, double expected) {
    cout << "testPower(" << base << ", " << exponent << "): " <<
flush;
    double result = power(base, exponent);
    cout << "\t" << result;
    if(result == expected) {
        cout << "\t[passed]" << endl;
    } else {
        cout << "\t[failed]" << endl;
    }
}
```

```cpp
#include <fstream>
#include <iostream>
#include <iomanip>
#include "console.h"
#include "timer.h"
#include "hashset.h"
#include "lexicon.h"
#include "queue.h"
#include "set.h"
#include "vector.h"
#include "grid.h"
#include "filelib.h"
#include "gwindow.h"
#include "gobjects.h"
#include "simpio.h"
#include "ghanoi.h"

using namespace std;

static const int N = 5;


void findSolution(int n, char source, char target, char aux) {
    // All about that base
    if(n == 1) {
        moveSingleDisk(source, target);
    // Recursive case
    } else {
        findSolution(n - 1, source, aux, target);
        moveSingleDisk(source, target);
        findSolution(n - 1, aux, target, source);
    }
}

int main() {
    cout << "Towers of Hanoi" << endl;
    initHanoiDisplay(N);
    findSolution(N, 'a', 'c', 'b');
    return 0;
}
```

```cpp
/*
 * File: ghanoi.cpp
 * ----------------
 * This file implements the graphical Hanoi functions.
 */

#include <iostream>
#include <string>
#include <memory> // For auto_ptr
#include "error.h"
#include "ghanoi.h"
#include "gobjects.h"
#include "gwindow.h"
#include "stack.h"
using namespace std;

/******** Constants ********/

/* Maximum permitted disks. */
const int kMaxDisks = 8;

/* Size of the window. */
const double kWindowWidth  = 500;
const double kWindowHeight = 300;

/* Number of spindles. */
const int kNumSpindles = 3;

/* Width of one spindle. */
const int kSpindleWidth = 20;

/* Pixel margin between spindle work areas. */
const double kSpindleMarginSize = 10;

/* Smallest possible disk width. */
const double kMinDiskWidth = 3 * kSpindleWidth;

/* Spindle color: Russet! */
const string kSpindleColor = "#80461B";

/* Colors for each disk. */
const string kDiskColors[] = {
   "Red",
   "Yellow",
   "Green",
   "Cyan",
   "Blue",
   "Magenta",
   "Orange",
   "Gray"
};
/* Border colors for each disk.  These are just darker versions of the disk colors
 * given above.
 */
const string kDiskBorderColors[] = {
   "#800000",
   "#808000",
   "#008000",
   "#008080",
   "#000080",
   "#800080",
   "#804000",
   "#404040"
};

/* Default pause time. */
const double kPauseTime = 500;

/* Number of steps in the animation. */
const int kNumAnimationSteps = 50;

struct Spindle {
    /* Which GRects are here right now. */
    Stack<GRect*> disksHere;

    /* The rectangle for this spindle. */
    GRect* rect;

    /* The start and end x coordinates of the area allocated
     * to this spindle.  This is the space where disks can be
     * moved.
     */
    double startX, endX;

    /* The center line for the spindle. */
    double centerX;
};
```

```
static struct HanoiGraphics {
    auto_ptr<GWindow> window;
    Vector<Spindle> spindles;

    int numDisks;
    double diskHeight;
} hg;

/******** Function Prototypes ********/

void setupSpindles();
void setupDisks();
double diskYPosition(int spindle);

/******** Implementations ********/

/* Initializes the display. */
void initHanoiDisplay(int numDisks) {
    /* Validate input. */
    if (numDisks > kMaxDisks) {
        error("Sorry, but we can't support that many disks.");
    }

    if (numDisks <= 0) {
        error("Sorry, but we need a positive number of disks.");
    }

    /* Create a new window to work with. */
    hg.window.reset(new GWindow(kWindowWidth, kWindowHeight));
    hg.window->setWindowTitle("Towers of Hanoi");

    /* Determine the height of a single disk.  We want to size the disks such that
     * each spindle can have all the disks on top of it, then some vertical space to
     * show the top of each spindle, then two blank spaces above it.  This works
     * out to needing a number of "virtual disks" equal to the total number of disks
     * actually used, plus four extras.
     */
    int numVirtualDisks = numDisks + 4;
    hg.diskHeight = hg.window->getHeight() / numVirtualDisks;
    hg.numDisks = numDisks;

    /* Initialize the spindles and disks. */
    setupSpindles();
    setupDisks();

    /* To let people see the setup, pause for a second before returning. */
    pause(kPauseTime);
}

/* Sets up the spindles. */
void setupSpindles() {
    /* Calculate the total horizontal area that can be allocated to a spindle. */
    double workspaceWidth = hg.window->getWidth() / kNumSpindles;

    for (int i = 0; i < kNumSpindles; i++) {
        Spindle spindle;

        /* Set the work area appropriately. */
        spindle.startX = workspaceWidth * i       + kSpindleMarginSize;
        spindle.endX   = workspaceWidth * (i + 1) - kSpindleMarginSize;

        /* Create a rectangle for this spindle.  The spindle will be centered in
         * the work area and will have height equal to the number of disks plus
         * one (so the spindle is still visible).  It will also be bottom-aligned.
         */
        double height = (hg.numDisks + 1) * hg.diskHeight;
        double y = hg.window->getHeight() - height;

        /* Determine where the center line is, then center the rectangle around that. */
        spindle.centerX = (spindle.startX + spindle.endX) / 2.0;
        spindle.rect = new GRect(spindle.centerX - kSpindleWidth / 2.0, y, kSpindleWidth, height);
        spindle.rect->setFilled(true);
        spindle.rect->setColor(kSpindleColor);

        /* Add that to the display. */
        hg.window->add(spindle.rect);

        /* Add this spindle to the list. */
        hg.spindles += spindle;
    }
}

/* Creates and sets up all of the disks that will be used in the simulation. */
void setupDisks() {
    for (int i = 0; i < hg.numDisks; i++) {
        /* We need to determine the position, color, and size of the disk.
```

```
         *
         * To size the disk, we will linearly interpolate between the workspace
         * area (the maximum possible width) and the minimum possible disk width
         * (specified as a constant).  In particular, we want the bottom disk to
         * have a size that perfectly fills the spindle workspace, and we want
         * the top disk to have size equal to kMinDiskWidth.  The formula we
         * will use for this is the following:
         *
         * Width of disk 0     = Workspace width.
         * Width of disk i - 1 = kMinDiskWidth.
         *
         * Therefore:
         *
         * width = ((kMinDiskWidth - workspaceWidth) / (numDisks - 1)) * i + workspaceWidth
         *
         * There is an edge case here when numDisks = 1, so we will special-case it.
         */
        double workspaceWidth = hg.spindles[0].endX - hg.spindles[0].startX;

        double width;
        if (hg.numDisks == 1) {
            width = workspaceWidth;
        } else {
            width = ((kMinDiskWidth - workspaceWidth) / (hg.numDisks - 1)) * i + workspaceWidth;
        }

        /* Given the width, the x coordinate can be found by taking the center line of
         * the spindle and backing off by half the width.
         */
        double x = hg.spindles[0].centerX - width / 2.0;

        /* We can determine the y coordinate of the disk by using our existing function
         * for determining where the next disk should go on a spindle.
         */
        double y = diskYPosition(0);

        /* Create the rectangle. */
        GRect* disk = new GRect(x, y, width, hg.diskHeight);
        disk->setColor(kDiskBorderColors[i]);
        disk->setFilled(true);
        disk->setFillColor(kDiskColors[i]);

        /* Draw the disk. */
        hg.window->add(disk);

        /* Add the disk to the spindle. */
        hg.spindles[0].disksHere.push(disk);
    }
}

/* Given a spindle number, returns the y coordinate where the next disk
 * would be placed on top of that spindle.
 */
double diskYPosition(int spindle) {
    if (spindle < 0 || spindle >= hg.spindles.size()) {
        error("Invalid spindle number.");
    }

    /* The position is determined as follows:
     *
     * 1. Start at the bottom of the window.
     * 2. Back off by the number of disks in the stack.
     * 3. Back off once more, since we need to give the upper y coordinate.
     *
     * This works out to windowHeight - (disksInStack + 1) * diskHeight.
     */
    return hg.window->getHeight() - (hg.spindles[spindle].disksHere.size() + 1) * hg.diskHeight;
}

/* Given a character, maps that character to a spindle number. */
int charToSpindle(char ch) {
    switch (ch) {
        case 'A': case 'a': return 0;
        case 'B': case 'b': return 1;
        case 'C': case 'c': return 2;
    }
    error("Unknown spindle.");
    return 0;
}

/* Given a time through the animation, represented by a number between 0 (start)
 * and 1 (end), interpolates where along the trajectory the disk would be at that
 * time using a cubic Hermitian spline.  This makes the animation look significantly
 * smoother than before.
 *
 * http://en.wikipedia.org/wiki/Cubic_Hermite_spline
 */
```

```
double interpolate(double t) {
    return -2 * t * t * t + 3 * t * t;
}

/* Animates a disk moving from its current position to the destination. */
void animateDiskPath(GRect* disk, double endX, double endY, double totalTime) {
    const double startX = disk->getX(), startY = disk->getY();

    /* Animate the motion! */
    for (int i = 0; i < kNumAnimationSteps; i++) {
        /* Interpolate between the start and end positions.  To determine
         * how much to interpolate, we use a cubic Hermite spline to map
         * from the fraction of the animation completed to a smoother
         * animation point.
         */
        double x = startX + (endX - startX) * interpolate(double(i) / (kNumAnimationSteps - 1));
        double y = startY + (endY - startY) * interpolate(double(i) / (kNumAnimationSteps - 1));
        disk->setLocation(x, y);

        /* Pause to let the animation progress.  This will not work out to pausing
         * for exactly the right amount of time because there's some latency involved
         * in the graphics calls, but it's "close enough."
         */
        pause(totalTime / kNumAnimationSteps);
    }
}

/* Animates a single disk moving from one spindle to another. */
void moveSingleDisk(char startCh, char finishCh) {
    /* Convert the start and end spindles to indices. */
    int start = charToSpindle(startCh), end = charToSpindle(finishCh);

    /* Confirm that the start spindle isn't empty. */
    if (hg.spindles[start].disksHere.isEmpty()) {
        error("No disks at the start spindle.");
    }

    /* Get the disk to move. */
    GRect* disk = hg.spindles[start].disksHere.pop();

    /* Make sure we can legally move the disk from the start spindle to the
     * end spindle.  This uses the size of the GRects as a proxy for the
     * size of the disk, which is probably not the best idea.  A better
     * implementation would use a struct to store both the GRect and its
     * logical size.
     */
    if (!hg.spindles[end].disksHere.isEmpty() &&
        hg.spindles[end].disksHere.top()->getWidth() < disk->getWidth()) {
        error("Cannot move a larger disk atop a smaller one.");
    }

    /* Determine how long the animation must take for each of the three parts of the
     * animation.
     */
    double eachAnimationTime = kPauseTime / 3;

    /* Move the disk up. */
    animateDiskPath(disk, disk->getX(), 0, eachAnimationTime);

    /* Move the disk over.  We need to compute the new x coordinate by centering
     * the disk over the new midline.
     */
    double newX = hg.spindles[end].centerX - disk->getWidth() / 2.0;
    animateDiskPath(disk, newX, 0, eachAnimationTime);

    /* Move the disk down. */
    animateDiskPath(disk, disk->getX(), diskYPosition(end), eachAnimationTime);

    /* Update internal state: make sure that the disk is now on the destination
     * spindle.
     */
    hg.spindles[end].disksHere.push(disk);
}
```