```
#include <iostream>
#include <ctime>
#include <string>

using namespace std;

class BooleanFunc
{
   static const int MAX_TABLE_FOR_CLASS = 65536;
   static const int DEFAULT_TABLE_SIZE = 16;

public:
   BooleanFunc(int tableSize = DEFAULT_TABLE_SIZE,
               bool evalReturnIfError = false);
   virtual ~BooleanFunc() { deAllocateTable(); }
   bool setTruthTableUsingTrue(int inputsThatProduceTrue[], int arraySize);
   bool setTruthTableUsingFalse(int inputsThatProduceFalse[], int arraySize);
   bool eval(int input);
   bool getState() const { return state; }

   BooleanFunc(const BooleanFunc& t);
   virtual BooleanFunc& operator =(const BooleanFunc& t);

private:
   int tableSize;
   bool *truthTable;
   bool evalReturnIfError;
   bool state;

   void setTableToConstant(bool constVal);
   bool inputInRange(int input);
   bool allocateTable(int numSags);
   void deAllocateTable();
};

class MultiSegmentLogic
{
   static const int DEFAULT_NUM_SEGS = 0;
protected:
   int numSegs;
   BooleanFunc *segs;

public:
   MultiSegmentLogic(int numSegs = DEFAULT_NUM_SEGS);
   virtual ~MultiSegmentLogic() { deAllocateSegs(); }
   bool setNumSegs(int numSegs);
   bool setSegment(int segNum, BooleanFunc &funcForThisSeg);
   void eval(int input);

   MultiSegmentLogic& operator =(const MultiSegmentLogic& t);
   MultiSegmentLogic(const MultiSegmentLogic& t);

protected:
   bool validSeg(int seg) const;
   bool allocateSegs(int numSegs);
   void deAllocateSegs();

};

class SevenSegmentLogic : public MultiSegmentLogic
{
public:
```

```
   SevenSegmentLogic();
   bool getValOfSeg(int seg) const;

private:
   //bool setSegment(int k, const BooleanFunc& bFunc);
   //void init();
   void loadAllFuncs();
};

class SevenSegmentImage
{
public:
   static const int MIN_HEIGHT = 5;
   static const int MAX_HEIGHT = 65;
   static const int MIN_WIDTH = 5;
   static const int MAX_WIDTH = 41;
   static const string DRAW_CHAR;
   static const string BLANK_CHAR;

private:
   bool **data;
   int topRow, midRow, bottomRow, leftCol, rightCol;

public:
   SevenSegmentImage(int width = MIN_WIDTH, int height = MIN_HEIGHT);
   ~SevenSegmentImage() { deallocateArray(); }
   void clearImage();
   bool turnOnCellsForSegment(char segment);
   bool setSize(int width, int height);
   void display() const;
   // deep copy stuff
   SevenSegmentImage(const SevenSegmentImage &tdi);
   const SevenSegmentImage& operator=(const SevenSegmentImage &rhs);
private:
   static bool validateSize(int width, int height);
   void allocateCleanArray();
   void deallocateArray();
};
const string SevenSegmentImage::DRAW_CHAR = "*";
const string SevenSegmentImage::BLANK_CHAR = " ";

class SevenSegmentDisplay
{
private:
   SevenSegmentImage theImage;
   SevenSegmentLogic theDisplay;
public:
   SevenSegmentDisplay
   (
    int width = SevenSegmentImage::MIN_WIDTH,
    int height = SevenSegmentImage::MIN_HEIGHT
    );
   bool setSize(int width, int height);
   void loadConsoleImage();
   void consoleDisplay() const;
   void eval(int input);
};


int main()
{
   SevenSegmentImage ssi;
```

```
   ssi.setSize(7, 9);
   ssi.turnOnCellsForSegment('a');
   ssi.display();
   ssi.turnOnCellsForSegment('b');
   ssi.display();
   ssi.turnOnCellsForSegment('c');
   ssi.display();
   ssi.turnOnCellsForSegment('d');
   ssi.display();
   ssi.clearImage();
   ssi.turnOnCellsForSegment('e');
   ssi.display();
   ssi.turnOnCellsForSegment('f');
   ssi.display();
   ssi.turnOnCellsForSegment('g');
   ssi.display();
   ssi.clearImage();
   ssi.turnOnCellsForSegment('x');
   ssi.display();
   ssi.turnOnCellsForSegment('3');
   ssi.display();

   SevenSegmentDisplay my7SegForCon(15, 13);
   my7SegForCon.setSize(5, 5);
   for (int j = 0; j < 16; j++) {
      my7SegForCon.eval(j);
      my7SegForCon.loadConsoleImage();
      my7SegForCon.consoleDisplay();
   }
   return 0;
}


BooleanFunc::BooleanFunc(int tableSize, bool evalReturnIfError) {
   // deal with construction errors in a crude but simple fashion
   if (tableSize > MAX_TABLE_FOR_CLASS || tableSize < 0)
      tableSize = DEFAULT_TABLE_SIZE;

   truthTable = NULL;
   allocateTable(tableSize);
   this->evalReturnIfError = evalReturnIfError;
   this->state = evalReturnIfError;
}
BooleanFunc& BooleanFunc::operator=(const BooleanFunc& that) {
   // always check this
   if (this == &that)
      return (*this);

   // reallocate table according to demands of "that." guaranteed to succeed
   allocateTable(that.tableSize);

   // copy the table to local
   for (int k = 0; k < tableSize; k++)
      truthTable[k] = that.truthTable[k];

   // set all non-table-related local private data
   state = that.state;
   evalReturnIfError = that.evalReturnIfError;

   return *this;
}
BooleanFunc::BooleanFunc(const BooleanFunc& that) {
```

```
   // let the overloaded assignment op do the work
   truthTable = NULL;
   *this = that;
}
bool BooleanFunc::setTruthTableUsingTrue(int inputsThatProduceTrue[],
                                         int arraySize)
{
   if (arraySize > tableSize)
      return false;
   for (int f = 0; f < tableSize; f++)
   {
      truthTable[f] = false;
   }

   for (int i = 0; i < arraySize; i++)
   {
      int t = inputsThatProduceTrue[i];
      if (t >= 0 && t < tableSize)
      {
         truthTable[t] = true;
      }
   }

   return true;
}
bool BooleanFunc::setTruthTableUsingFalse(int inputsThatProduceFalse[],
                                          int arraySize)
{
   if (arraySize > tableSize)
      return false;
   for (int t = 0; t < tableSize; t++)
   {
      truthTable[t] = true;
   }

   for (int f = 0; f < arraySize; f++)
   {
      int t = inputsThatProduceFalse[f];
      if (t >= 0 && t < tableSize)
      {
         truthTable[t] = false;
      }
   }
   return true;
}

bool BooleanFunc::eval(int input) {
   if (!inputInRange(input))
      return (state = evalReturnIfError);
   return (state = truthTable[input]);
}

void BooleanFunc::setTableToConstant(bool constVal) {
   for (int k = 0; k < tableSize; k++)
      truthTable[k] = constVal;
}
bool BooleanFunc::inputInRange(int input) {
   return (input >= 0 && input < tableSize);
}
void BooleanFunc::deAllocateTable() {
   if (truthTable)
      delete[] truthTable;
```

```
   truthTable = NULL;
   tableSize = 0;
}
bool BooleanFunc::allocateTable(int tableSize) {
   if (tableSize < 1 || tableSize > MAX_TABLE_FOR_CLASS)
      return false;

   deAllocateTable();
   truthTable = new bool[tableSize];
   this->tableSize = tableSize;

   setTableToConstant(false);
   return true;
}

MultiSegmentLogic::MultiSegmentLogic(int numSegs) {
   segs = NULL;
   if (!allocateSegs(numSegs))
      allocateSegs(DEFAULT_NUM_SEGS);
}

MultiSegmentLogic::MultiSegmentLogic(const MultiSegmentLogic& that) {

   *this = that;
}
MultiSegmentLogic& MultiSegmentLogic::operator=(const MultiSegmentLogic& that) {
   if (this == &that)
      return *this;

   allocateSegs(that.numSegs);

   for (int k = 0; k < numSegs; k++)
      segs[k] = that.segs[k];

   return *this;
}

bool MultiSegmentLogic::setNumSegs(int numSegs) {
   return allocateSegs(numSegs);
}
bool MultiSegmentLogic::setSegment(int segNum, BooleanFunc& funcForThisSeg) {
   if (!validSeg(segNum))
      return false;

   segs[segNum] = funcForThisSeg;

   return true;
}

bool MultiSegmentLogic::validSeg(int seg) const {
   return (seg >= 0 && seg < numSegs);
}
void MultiSegmentLogic::eval(int input) {
   for (int k = 0; k < numSegs; k++)
      segs[k].eval(input);
}
void MultiSegmentLogic::deAllocateSegs() {
   if (segs != NULL)
      delete[] segs;
   segs = NULL;
   numSegs = 0;
}
```

we blend it's consistency

```
bool MultiSegmentLogic::allocateSegs(int numSegs) {
   if (numSegs < 0)
      return false;

   deAllocateSegs();
   segs = new BooleanFunc[numSegs];
   this->numSegs = numSegs;
   return true;
}

SevenSegmentLogic::SevenSegmentLogic() : MultiSegmentLogic(7) {
   loadAllFuncs();
}
bool SevenSegmentLogic::getValOfSeg(int seg) const {
   if (!validSeg(seg))
      return false;
   return segs[seg].getState();
}
void SevenSegmentLogic::loadAllFuncs() {

   static BooleanFunc a(16, true);
   static BooleanFunc b(16, false);
   static BooleanFunc c(16, false);
   static BooleanFunc d(16, true);
   static BooleanFunc e(16, true);
   static BooleanFunc f(16, true);
   static BooleanFunc g(16, true);
   static bool funcsAlreadyDefined = false;

   if (!funcsAlreadyDefined) {

      static int segA[] = { 1, 4, 11, 13 };
      static int segB[] = { 5, 6, 11, 12, 14, 15 };
      static int segC[] = { 2, 12, 14, 15 };
      static int segD[] = { 1, 4, 7, 10, 15 };
      static int segE[] = { 1, 3, 4, 5, 7, 9 };
      static int segF[] = { 1, 2, 3, 7, 13 };
      static int segG[] = { 0, 1, 7, 12 };

      a.setTruthTableUsingFalse(segA, sizeof(segA) / sizeof(int));
      b.setTruthTableUsingFalse(segB, sizeof(segB) / sizeof(int));
      c.setTruthTableUsingFalse(segC, sizeof(segC) / sizeof(int));
      d.setTruthTableUsingFalse(segD, sizeof(segD) / sizeof(int));
      e.setTruthTableUsingFalse(segE, sizeof(segE) / sizeof(int));
      f.setTruthTableUsingFalse(segF, sizeof(segF) / sizeof(int));
      g.setTruthTableUsingFalse(segG, sizeof(segG) / sizeof(int));

      funcsAlreadyDefined = true;
   }

   // this block loads the data for this particular object
   setSegment(0, a);
   setSegment(1, b);
   setSegment(2, c);
   setSegment(3, d);
   setSegment(4, e);
   setSegment(5, f);
   setSegment(6, g);
}

SevenSegmentImage::SevenSegmentImage(int width, int height)
```

```
{
   this->data = NULL;
   setSize(width, height);
}

void SevenSegmentImage::clearImage()
{
   for (int row = topRow; row <= bottomRow; row++)
   {
      for (int col = leftCol; col <= rightCol; col++)
      {
         data[row][col] = false;
      }
   }
}

bool SevenSegmentImage::turnOnCellsForSegment(char segment)
{
   if (segment >= 'a' && segment <= 'g')
   {
      if (segment == 'a' || segment == 'A')
      {
         for (int i = leftCol; i <= rightCol; i++)
         {
            data[topRow][i] = true;
         }
      }
      else if (segment == 'b' || segment == 'B')
      {
         for (int i = topRow; i <= midRow; i++)
         {
            data[i][rightCol] = true;
         }
      }
      else if (segment == 'c' || segment == 'C')
      {
         for (int i = midRow; i <= bottomRow; i++)
         {
            data[i][rightCol] = true;
         }
      }
      else if (segment == 'd' || segment == 'D')
      {
         for (int i = leftCol; i <= rightCol; i++)
         {
            data[bottomRow][i] = true;
         }
      }
      else if (segment == 'e' || segment == 'E')
      {
         for (int i = midRow; i <= bottomRow; i++)
         {
            data[i][leftCol] = true;
         }
      }
      else if (segment == 'f' || segment == 'F')
      {
         for (int i = topRow; i <= midRow; i++)
         {
            data[i][leftCol] = true;
         }
      }
```

```
      else if (segment == 'g' || segment == 'G')
      {
         for (int i = leftCol; i <= rightCol; i++)
         {
            data[midRow][i] = true;
         }
      }
      return true;
   }
   return false;
}

bool SevenSegmentImage::setSize(int width, int height)
{

   if (validateSize(width, height))
   {
      deallocateArray();
      topRow = 0;
      midRow = height / 2;
      bottomRow = height - 1;
      leftCol = 0;
      rightCol = width - 1;

      allocateCleanArray();
      return true;
   }
   else
   {
      return false;
   }
}

void SevenSegmentImage::display() const
{
   for (int row = topRow; row <= bottomRow; row++)
   {
      for (int col = leftCol; col <= rightCol; col++)
      {
         if (data[row][col])
            cout << DRAW_CHAR;
         else
            cout << BLANK_CHAR;

      }
      cout << "\n";
   }

   cout << "\n";
}

// deep copy stuff
SevenSegmentImage::SevenSegmentImage(const SevenSegmentImage &tdi)
{
   data = NULL;
   *this = tdi;
}

const SevenSegmentImage& SevenSegmentImage::operator=(const SevenSegmentImage &r
hs)
{
   // always check this
```

```
   if (this == &rhs)
      return (*this);

   if (this != &rhs)
   {
      this->deallocateArray();
      this->topRow = rhs.topRow;
      this->midRow = rhs.midRow;
      this->bottomRow = rhs.bottomRow;
      this->leftCol = rhs.leftCol;
      this->rightCol = rhs.rightCol;

      allocateCleanArray();

      for (int row = topRow; row < rhs.bottomRow; row++)
      {
         this->data[row] = new bool[rightCol + 1];
         for (int col = topRow; col < rhs.rightCol; col++)
            data[row][col] = rhs.data[row][col];
      }
   }

   return *this;
}

bool SevenSegmentImage::validateSize(int width, int height)
{
   return (width <= MAX_WIDTH && width >= MIN_WIDTH && height >= MIN_HEIGHT &&
           height <= MAX_HEIGHT && width % 2 != 0 && height % 2 != 0);
}

void SevenSegmentImage::allocateCleanArray()
{
   this->data = new bool*[bottomRow + 1];
   for (int row = topRow; row <= bottomRow; row++)
   {
      this->data[row] = new bool[rightCol + 1];
   }
   clearImage();
}

void SevenSegmentImage::deallocateArray()
{
   if (this->data == NULL)
      return;

   for (int i = topRow; i <= bottomRow; i++)
   {
      delete[] this->data[i];
   }
   delete[] this->data;
   this->data = NULL;
}

SevenSegmentDisplay::SevenSegmentDisplay(int width, int height)
{
   theDisplay = SevenSegmentLogic();
   if(width >= SevenSegmentImage::MIN_WIDTH
      && width <= SevenSegmentImage::MAX_WIDTH
      && height >= SevenSegmentImage::MIN_HEIGHT
      && height <= SevenSegmentImage::MAX_HEIGHT)
   {
```

```
      theImage = SevenSegmentImage(width, height);
   }
   else
   {
      theImage = SevenSegmentImage(SevenSegmentImage::MIN_WIDTH,
                                   SevenSegmentImage::MIN_HEIGHT);
   }
}

bool SevenSegmentDisplay::setSize(int width, int height)
{
   if(width >= SevenSegmentImage::MIN_WIDTH
      && width <= SevenSegmentImage::MAX_WIDTH
      && height >= SevenSegmentImage::MIN_HEIGHT
      && height <= SevenSegmentImage::MAX_HEIGHT)
   {
      theImage.setSize(width, height);
      return true;
   }
   else
   {
      return false;
   }
}

void SevenSegmentDisplay::loadConsoleImage()
{
   theImage.clearImage();
   for (int k = 0; k < 7; k++)
   {
      if(theDisplay.getValOfSeg(k))
      {
         char c = 'a' + k;
         theImage.turnOnCellsForSegment(c);
      }
   }
}

void SevenSegmentDisplay::consoleDisplay() const
{
   theImage.display();
}

void SevenSegmentDisplay::eval(int input)
{
   for (int k = 0; k < 7; k++)
   {
      theDisplay.eval(input);
   }
}

/*---------------------------- Posted Run ----------------------------------
-
*******
```

****
*
*
*
*

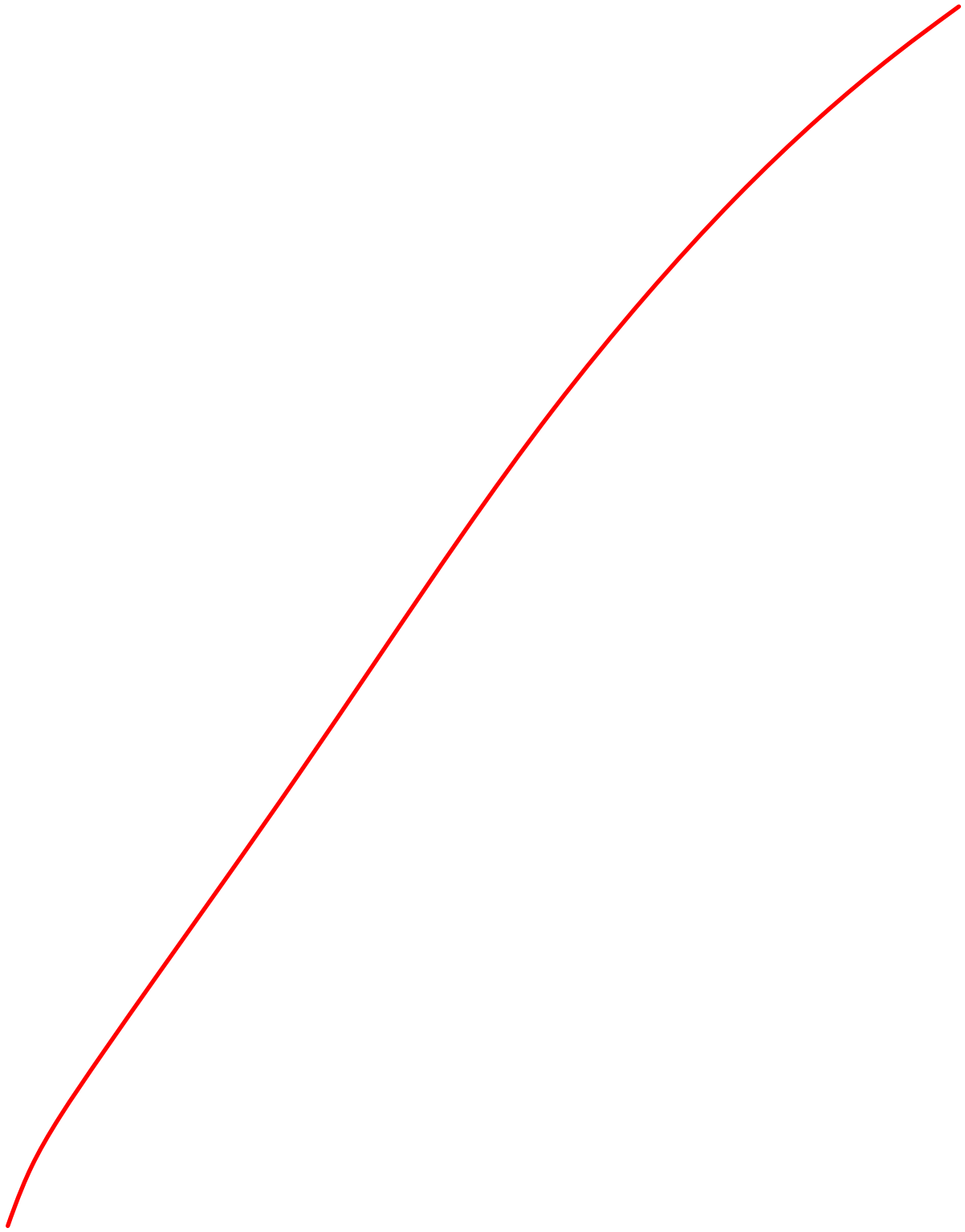
****
*
*
*
*
*
*
*
*

****
*
*
*
*
*
*
*
****

*
*
*
*
*

*
*
*
*
*
*
*
*
*

*
*
*
*
****
*
*
*
*



*****
*   *
*   *
*   *
*****

*
*
*
*
*

*****
*
*****
*
*****

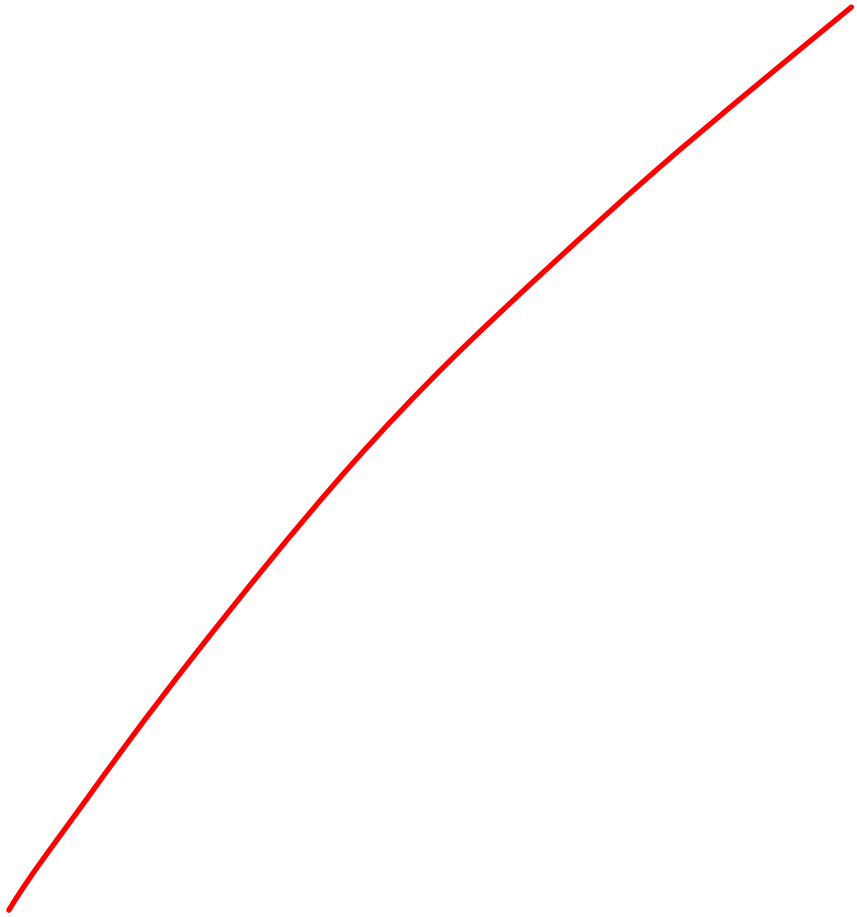
*****
*
*****
*
*****

*   *
*   *
*****
*
*

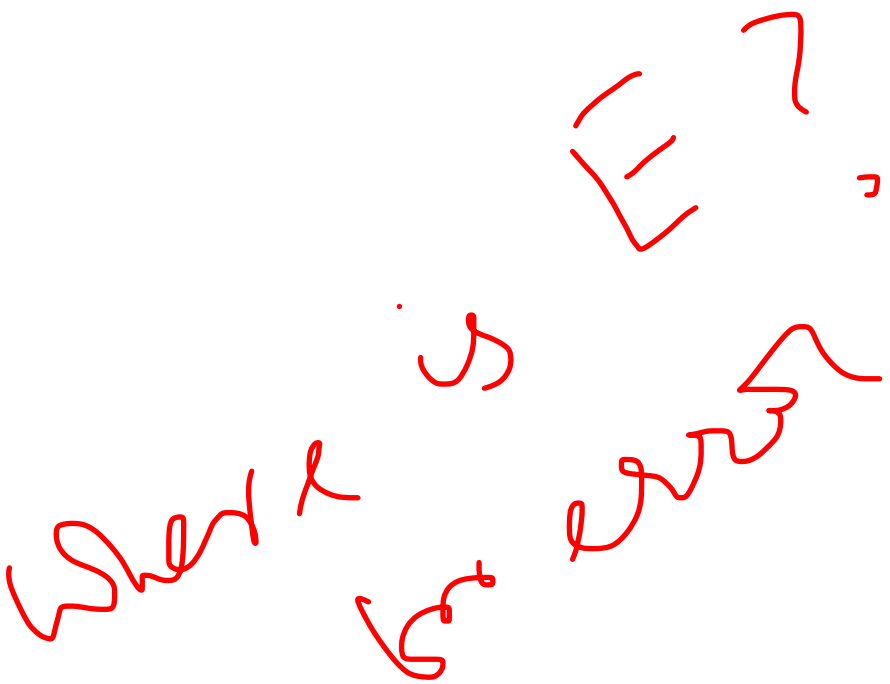
*****
*
*****
*
*****

*****
*
*****
*   *
*****

*****



```
*
*
*
*

*****
*   *
*****
*   *
*****

*****
*   *
*****
*
*****

*****
*   *
*****
*   *
*   *

*
*
*****
*   *
*****

*****
*
*
*
*****

*
*
*****
*   *
*****

*****
*
*****
*
*****

*****
*
*****
*
*
```



```
Press any key to continue . . .
--------------------------------------------------------------------------------
-*/
```

```
// CS 2B Lab 7
// Instructor Solution:
//  Original - Prof. Loceff, Updates, Edits, Annotations:&

// Notes:
//  - Use of sensible names for vars
//  - Correct Boolean logic
//  - Correct determination of middle row
//  - No out of bounds access
//  - Faithfulness to spec
//  - Correct method qualifications (including virtuals)

#include <iostream>
#include <ctime>
#include <string>

using namespace std;

// ----- BooleanFunc -----

class BooleanFunc {
    static const int MAX_TABLE_FOR_CLASS = 65536;  // that's 16 binary inputs
    static const int DEFAULT_TABLE_SIZE = 16;

private:
    int tableSize;
    bool *truthTable;
    bool evalReturnIfError;
    bool state;

public:
    BooleanFunc(int tSize = DEFAULT_TABLE_SIZE, bool evalReturnIfError = false);
    virtual ~BooleanFunc() { deAllocateTable(); }

    bool setTruthTableUsingTrue(int inputsThatProduceTrue[], int arraySize);
    bool setTruthTableUsingFalse(int inputsThatProduceFalse[], int arraySize);
    bool eval(int input);

    bool getState() const { return state; }

    // deep copy required methods
    BooleanFunc(const BooleanFunc& that);
    virtual BooleanFunc& operator=(const BooleanFunc& that);

private:
    // helpers
    void setTableToConstant(bool constVal);
    bool inputInRange(int input);
    bool allocateTable(int numSegs);
    void deAllocateTable();
};

BooleanFunc::BooleanFunc(int tableSize, bool evalReturnIfError) {
    // deal with construction errors in a crude but simple fashion
    if (tableSize > MAX_TABLE_FOR_CLASS || tableSize < 0)
        tableSize = DEFAULT_TABLE_SIZE;

    truthTable = NULL;
    allocateTable(tableSize);
    this->evalReturnIfError = evalReturnIfError;
    this->state = evalReturnIfError;
}

BooleanFunc& BooleanFunc::operator=(const BooleanFunc& that) {
```

```
    // always check this
    if (this == &that)
        return (*this);

    // reallocate table according to demands of "that." guaranteed to succeed
    allocateTable(that.tableSize);

    // copy the table to local
    for (int k = 0; k < tableSize; k++)
        truthTable[k] = that.truthTable[k];

    // set all non-table-related local private data
    state = that.state;
    evalReturnIfError = that.evalReturnIfError;

    return *this;
}

BooleanFunc::BooleanFunc(const BooleanFunc& that) {
    // let the overloaded assignment op do the work
    truthTable = NULL;
    *this = that;
}

bool BooleanFunc::setTruthTableUsingTrue(int *inputsThatProduceTrue,
                                         int arraySize) {
    if (arraySize > tableSize) return false;

    // they are giving us true values, so we init to false then overwrite
    setTableToConstant(false);

    for (int k = 0; k < arraySize; k++) {
        int kTable = inputsThatProduceTrue[k];
        if (kTable >= 0 && kTable < tableSize)
            truthTable[kTable] = true;
    }

    return true;
}

bool BooleanFunc::setTruthTableUsingFalse(int *inputsThatProduceFalse,
                                          int arraySize) {
    if (arraySize > tableSize) return false;

    // they are giving us false values, so we init to true then overwrite
    setTableToConstant(true);

    for (int k = 0; k < arraySize; k++) {
        int kTable = inputsThatProduceFalse[k];
        if (kTable >= 0 && kTable < tableSize)
            truthTable[kTable] = false;
    }

    return true;
}

// Can't be a const method because it sets state
bool BooleanFunc::eval(int input) {
    if (!inputInRange(input))
        return (state = evalReturnIfError);
    return (state = truthTable[input]);
}
```

```
// private  helpers
void BooleanFunc::setTableToConstant(bool constVal) {
    for (int k = 0; k < tableSize; k++)
        truthTable[k] = constVal;
}

bool BooleanFunc::inputInRange(int input) {
    return (input >= 0 && input < tableSize);
}

void BooleanFunc::deAllocateTable() {
    if (truthTable)
        delete[] truthTable;
    truthTable = NULL;
    tableSize = 0;
}

bool BooleanFunc::allocateTable(int tableSize) {
    if (tableSize < 1 || tableSize > MAX_TABLE_FOR_CLASS)
        return false;

    deAllocateTable();
    truthTable = new bool[tableSize];
    this->tableSize = tableSize;

    // so we have a default function - identically 0;
    setTableToConstant(false);
    return true;
}

// ----- MultiSegmentLogic -----

class MultiSegmentLogic {
    static const int DEFAULT_NUM_SEGS = 0;

protected:
    BooleanFunc *segs;
    int numSegs;

public:
    MultiSegmentLogic(int numSegs = DEFAULT_NUM_SEGS);
    virtual ~MultiSegmentLogic() { deAllocateSegs(); }

    bool setNumSegs(int numSegs);
    bool setSegment(int segNum, BooleanFunc& funcForThisSeg);
    void eval(int input);

    // deep copy required methods
    MultiSegmentLogic(const MultiSegmentLogic& that);
    virtual MultiSegmentLogic& operator=(const MultiSegmentLogic& that);

protected:
    // helpers
    bool validSeg(int seg) const;
    bool allocateSegs(int numSegs);
    void deAllocateSegs();
};

MultiSegmentLogic::MultiSegmentLogic(int numSegs) {
    segs = NULL;  // needed for mutator
    if (!allocateSegs(numSegs))
        allocateSegs(DEFAULT_NUM_SEGS);
}
```

```
// copy constructor and assignment operator
MultiSegmentLogic::MultiSegmentLogic(const MultiSegmentLogic& that) {
    // let the overloaded assignment op do the work
    *this = that;
}

MultiSegmentLogic& MultiSegmentLogic::operator=(const MultiSegmentLogic& that) {
    // always check this
    if (this == &that)
        return *this;

    // reallocate according to demands of "that." guaranteed to succeed
    allocateSegs(that.numSegs);

    // copy the segments to local (note that BooleanFunc's overloaded = implied
    for (int k = 0; k < numSegs; k++)
        segs[k] = that.segs[k];

    return *this;
}

// allow this public to call private even though nothing added for future use
bool MultiSegmentLogic::setNumSegs(int numSegs) {
    return allocateSegs(numSegs);
}

bool MultiSegmentLogic::setSegment(int segNum, BooleanFunc& funcForThisSeg) {
    if (!validSeg(segNum))
        return false;

    // assignment copies object so we can pass in anon/temporary BooleanFunc
    segs[segNum] = funcForThisSeg;

    return true;
}

// private helpers
bool MultiSegmentLogic::validSeg(int seg) const {
    return (seg >= 0 && seg < numSegs);
}

void MultiSegmentLogic::eval(int input) {
    for (int k = 0; k < numSegs; k++)
        segs[k].eval(input);
}

void MultiSegmentLogic::deAllocateSegs() {
    if (segs != NULL)
        delete[] segs;
    segs = NULL;
    numSegs = 0;
}

// could be eliminated and everything put into setNumSegs(), but has symmetry
bool MultiSegmentLogic::allocateSegs(int numSegs) {
    if (numSegs < 0)
        return false;

    deAllocateSegs();
    segs = new BooleanFunc[numSegs];
    this->numSegs = numSegs;
    return true;
}
```

```
// ----- SevenSegmentLogic -----

class SevenSegmentLogic : public MultiSegmentLogic {
public:
    SevenSegmentLogic();
    bool getValOfSeg(int seg) const;

private:
    void loadAllFuncs();
};

// Note: 7 is not a magic number, cuz it's a... duh... SEVEN segment display
SevenSegmentLogic::SevenSegmentLogic() : MultiSegmentLogic(7) {
    loadAllFuncs();
}

bool SevenSegmentLogic::getValOfSeg(int seg) const {
    if (!validSeg(seg))
        return false;
    return segs[seg].getState();
}

void SevenSegmentLogic::loadAllFuncs() {
    // we use letters, rather than arrays, to help connect with traditional
    // a - g segements and make every step crystal clear

    // set error pattern to "E" through second parameter
    // these must be static since they are only needed once, ever and this
    // avoids reinstantiation in multiple objects
    static BooleanFunc a(16, true);
    static BooleanFunc b(16, false);
    static BooleanFunc c(16, false);
    static BooleanFunc d(16, true);
    static BooleanFunc e(16, true);
    static BooleanFunc f(16, true);
    static BooleanFunc g(16, true);
    static bool funcsAlreadyDefined = false;

    // we only need to define these arrays and BooleanFuncs once per program
    if (!funcsAlreadyDefined) {
        // define in terms of on/true
        // (can remove static to impr. storage effieciency)
        static int segA[] = { 1, 4, 11, 13 };
        static int segB[] = { 5, 6, 11, 12, 14, 15 };
        static int segC[] = { 2, 12, 14, 15 };
        static int segD[] = { 1, 4, 7, 10, 15 };
        static int segE[] = { 1, 3, 4, 5, 7, 9 };
        static int segF[] = { 1, 2, 3, 7, 13 };
        static int segG[] = { 0, 1, 7, 12 };

        a.setTruthTableUsingFalse(segA, sizeof(segA) / sizeof(int));
        b.setTruthTableUsingFalse(segB, sizeof(segB) / sizeof(int));
        c.setTruthTableUsingFalse(segC, sizeof(segC) / sizeof(int));
        d.setTruthTableUsingFalse(segD, sizeof(segD) / sizeof(int));
        e.setTruthTableUsingFalse(segE, sizeof(segE) / sizeof(int));
        f.setTruthTableUsingFalse(segF, sizeof(segF) / sizeof(int));
        g.setTruthTableUsingFalse(segG, sizeof(segG) / sizeof(int));

        funcsAlreadyDefined = true;
    }

    // this block loads the data for this particular object
    setSegment(0, a);
    setSegment(1, b);
```

```
    setSegment(2, c);
    setSegment(3, d);
    setSegment(4, e);
    setSegment(5, f);
    setSegment(6, g);
}

// ----- SevenSegmentImage -----

class SevenSegmentImage {
public:
    static const int MIN_HEIGHT = 5;
    static const int MIN_WIDTH = 5;
    static const int MAX_HEIGHT = 65;
    static const int MAX_WIDTH = 41;
    static const string DRAW_CHAR;
    static const string BLANK_CHAR;

private:
    bool **data;
    int topRow, midRow, bottomRow, leftCol, rightCol;

public:
    SevenSegmentImage(int width = MIN_WIDTH, int height = MIN_HEIGHT);
    ~SevenSegmentImage() { deallocateArray(); }

    void clearImage();
    bool turnOnCellsForSegment(char segment);
    bool setSize(int width, int height);

    void display() const;

    // deep copy stuff
    SevenSegmentImage(const SevenSegmentImage &tdi);
    const SevenSegmentImage &operator=(const SevenSegmentImage &rhs);

private:
    static bool validateSize(int width, int height);
    void allocateCleanArray();
    void deallocateArray();

    // helpers - not required, but used by instructor
    void drawHorizontal(int row);
    void drawVertical(int col, int startRow, int stopRow);
};

const string SevenSegmentImage::DRAW_CHAR = "*";
const string SevenSegmentImage::BLANK_CHAR = " ";

SevenSegmentImage::SevenSegmentImage(int width, int height) {
    // needed by setSize() which calls allocate().
    data = NULL;

    if (!setSize( width, height))
        setSize(MIN_HEIGHT, MIN_WIDTH);
}

// sets size, allocates memory (deletes old memory)
bool SevenSegmentImage::setSize(int width, int height) {
    if (!validateSize(width, height))
        return false;

    // done first, since it relies on old topRow, etc.
    deallocateArray();
```

```
    // even though bottom and left are 0, we use variables for clarity
    topRow = height - 1;
    midRow = height / 2;  // correct:  odd# gives middle
    bottomRow = 0;

    rightCol = width - 1;
    leftCol = 0;

    allocateCleanArray();
    return true;
}

void SevenSegmentImage::clearImage() {
    int row, col;
    int height, width;

    height = topRow + 1;
    width = rightCol + 1;

    for (row = 0; row < height; row++) {
        for (col = 0; col < width; col++)
            data[row][col] = false;
    }
}

bool SevenSegmentImage::turnOnCellsForSegment(char segment) {
    char displayLetter;

    displayLetter = tolower( segment );

    switch (displayLetter) {
        case 'a':
            drawHorizontal(topRow);
            break;
        case 'g':
            drawHorizontal(midRow);
            break;
        case 'd':
            drawHorizontal(bottomRow);
            break;
        case 'e':
            drawVertical(leftCol, bottomRow, midRow);
            break;
        case 'f':
            drawVertical(leftCol, midRow, topRow);
            break;
        case 'b':
            drawVertical(rightCol, midRow, topRow);
            break;
        case 'c':
            drawVertical(rightCol, bottomRow, midRow);
            break;
        default:
            // out-of-range
            return false;
    }
    return true;
}

void SevenSegmentImage::drawHorizontal(int row) {
    for (int k = leftCol; k <= rightCol; k++)
        data[row][k] = true;
}
```

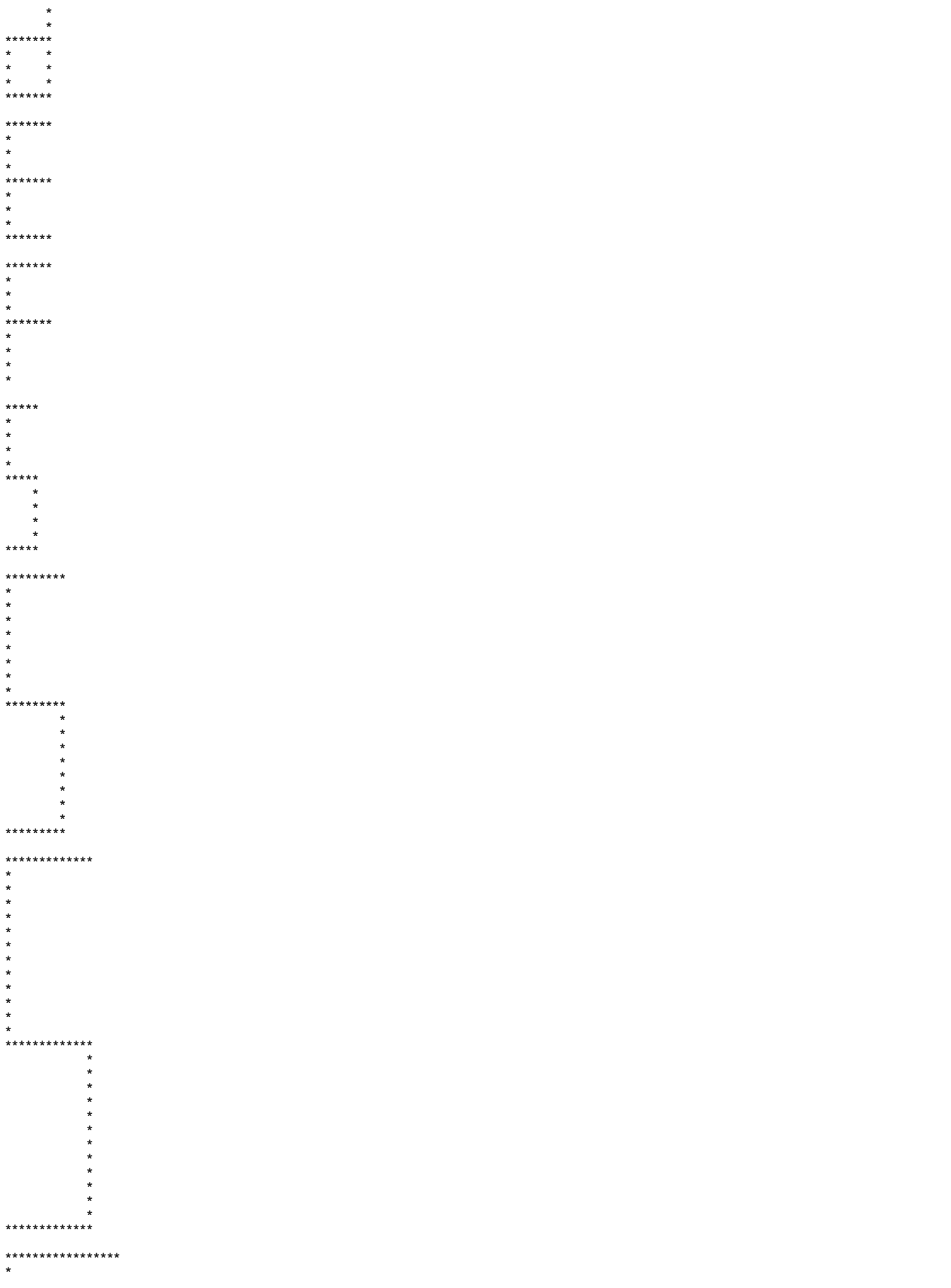
```
void SevenSegmentImage::drawVertical(int col, int startRow, int stopRow) {
    for (int k = startRow; k <= stopRow; k++)
        data[k][col] = true;
}

bool SevenSegmentImage::validateSize(int width, int height) {
    // must be in range and also odd
    if (width > MAX_WIDTH || width < MIN_WIDTH)
        return false;
    if (height > MAX_HEIGHT || height < MIN_HEIGHT || height % 2 == 0)
        return false;
    return true;
}

SevenSegmentImage::SevenSegmentImage(const SevenSegmentImage &other) {
    data = NULL;    // in prep for downstream dealloc()
    *this = other;
}

const SevenSegmentImage &SevenSegmentImage::
                        operator=(const SevenSegmentImage &that) {
    if (this == &that)
        return (*this);

    //  does the hard work
    setSize(that.rightCol + 1, that.topRow + 1 );

    // copies image data
    for (int row = topRow; row >= bottomRow; row--)
        for (int col = leftCol; col <= rightCol; col++)
            data[row][col] = that.data[row][col];

    return *this;
}

void SevenSegmentImage::display() const {
    cout << endl;
    for (int row = topRow; row >= bottomRow; row--) {
        for (int col = leftCol; col <= rightCol; col++)
            cout << (data[row][col] ? DRAW_CHAR : BLANK_CHAR);
        cout << endl;
    }
}

// this approach requires prior dealloc, and topRow, rightCol defined
void SevenSegmentImage::allocateCleanArray() {
    int height = topRow + 1;
    int width = rightCol + 1;

    // massive error. should be impossible coming in.
    if (data != NULL)
        return;

    data = new bool*[height];
    for (int row = 0; row < height; row++)
        data[row] = new bool[width];

    clearImage();
}

void SevenSegmentImage::deallocateArray() {
    if (data == NULL)
        return;
```

```
    int height = topRow + 1;

    for (int row = 0; row < height; row++)
        delete data[row];

    delete[] data;
    data = NULL;
}
// ----------------------------------------------------------------------------

class SevenSegmentDisplay {
private:
    SevenSegmentImage theImage;
    SevenSegmentLogic theDisplay;

public:
    SevenSegmentDisplay(int width = SevenSegmentImage::MIN_WIDTH,
                        int height = SevenSegmentImage::MIN_HEIGHT);

    bool setSize(int width, int height);
    void loadConsoleImage();
    void consoleDisplay() const;
    void eval(int input);
};

SevenSegmentDisplay::SevenSegmentDisplay(int width, int height) {
    theImage.setSize(width, height);
}

bool SevenSegmentDisplay::setSize(int width, int height) {
    return theImage.setSize(width, height);
}

void SevenSegmentDisplay::eval(int input) {
    theDisplay.eval(input);
}

void SevenSegmentDisplay::loadConsoleImage() {
    char convertIntToSeg[] = { 'a', 'b', 'c', 'd', 'e', 'f', 'g' };

    theImage.clearImage();
    for (int k = 0; k < 7; k++) {
        if (theDisplay.getValOfSeg(k))
            theImage.turnOnCellsForSegment(convertIntToSeg[k]);
    }
}

void SevenSegmentDisplay::consoleDisplay() const {
    theImage.display();
}

// ----- Main Test Driver -----
// main client ----------------------------------------------------------
int main()
{
    SevenSegmentImage ssi;
    SevenSegmentDisplay  my7SegForCon(15, 13);

    cout << "----- Testing SevenSegmentImage -----\n";

    ssi.setSize(7, 9);
    ssi.turnOnCellsForSegment('a');  ssi.display();
    ssi.turnOnCellsForSegment('b');  ssi.display();
```

```
    ssi.turnOnCellsForSegment('c');  ssi.display();
    ssi.turnOnCellsForSegment('d');  ssi.display();

    ssi.clearImage();
    ssi.turnOnCellsForSegment('e');  ssi.display();
    ssi.turnOnCellsForSegment('f');  ssi.display();
    ssi.turnOnCellsForSegment('g');  ssi.display();

    ssi.clearImage();
    ssi.turnOnCellsForSegment('x');  ssi.display();
    ssi.turnOnCellsForSegment('3');  ssi.display();

    cout << "----- Testing SevenSegmentDisplay -----\n";

    my7SegForCon.setSize(7, 9);
    for (int j = 0; j < 16; j++) {
        my7SegForCon.eval(j);
        my7SegForCon.loadConsoleImage();
        my7SegForCon.consoleDisplay();
    }

    for (int j = 5; j < 21; j += 4) {
        my7SegForCon.setSize(j, 2*j + 1);
        my7SegForCon.eval(5);
        my7SegForCon.loadConsoleImage();
        my7SegForCon.consoleDisplay();
    }

    return 0;
}

/* ----- Test Runs -----

*******

*******
     *
     *
     *
     *

*******
     *
     *
     *
     *
     *
     *
     *
     *

*******
     *
     *
     *
     *
     *
     *
     *
*******
```

```
*
*
*
*
*

*
*
*
*
*
*
*
*
*

*
*
*
*
*******
*
*
*
*
```

```
----- Testing SevenSegmentDisplay -----

*******
*     *
*     *
*     *
*     *
*     *
*     *
*     *
*******

      *
      *
      *
      *
      *
      *
      *
      *
      *

*******
      *
      *
      *
*******
*
*
*
*******

*******
      *
      *
      *
*******
      *
      *
      *
*******

*     *
*     *
```



```
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
**********
          *
          *
          *
          *
          *
          *
          *
          *
          *
          *
          *
          *
          *
          *
          *
**********
Program ended with exit code: 0
Press any key to continue . . .
------------------------------------------------------------------------ */
```