```
#include <iostream>

using namespace std;

class BooleanFunc
{
public:
    BooleanFunc(int tableSize = DEFAULT_TABLE_SIZE, bool evalReturnIfError = fals
e);
    BooleanFunc(const BooleanFunc& t);
    ~BooleanFunc();
    bool setTruthTableUsingTrue(int inputsThatProduceTrue[], int arraySize);
    bool setTruthTableUsingFalse(int inputsThatProduceFalse[], int arraySize);
    bool eval(int input);
    bool getState() { return state; }
    static const int MAX_TABLE_FOR_CLASS = 65536;
    static const int DEFAULT_TABLE_SIZE = 16;
    BooleanFunc& operator =(const BooleanFunc& t);

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

class MultiSegmentLogic
{
public:
    MultiSegmentLogic(int numSegs = 0);
    ~MultiSegmentLogic();
    bool setNumSegs(int numSegs);
    bool setSegment(int segNum, BooleanFunc &funcForThisSeg);
    void eval(int input);
    MultiSegmentLogic& operator =(const MultiSegmentLogic& t);
    MultiSegmentLogic(const MultiSegmentLogic& t);

private:

protected:
    int numSegs;
    BooleanFunc *segs;

};

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

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

int main()
{
    BooleanFunc segA, segB(13), segC(100, true), segD, segF;
    int evenFunc[] = { 0, 2, 4, 6, 8, 10, 12, 14 }, inputX;
    short sizeEvenFunc = sizeof(evenFunc) / sizeof(evenFunc[0]);
    int greater9Func[] = { 10, 11, 12, 13, 14, 15 };
```

```
    short sizeGreater9Func = sizeof(greater9Func) / sizeof(greater9Func[0]);
    int greater3Func[] = { 0, 1, 2, 3 };
    short sizeGreater3Func = sizeof(greater3Func) / sizeof(greater3Func[0]);
    segA.setTruthTableUsingTrue(evenFunc, sizeEvenFunc);
    segB.setTruthTableUsingTrue(greater9Func, sizeGreater9Func);
    segC.setTruthTableUsingFalse(greater3Func, sizeGreater3Func);
    segD = segA;
    segF = BooleanFunc(segC);
    // testing class BooleanFunc
    cout << "before eval()\n";
    cout
        << "\n A(x) = "
        << segA.getState()
        << "\n B(x) = "
        << segB.getState()
        << "\n C(x) = "
        << segC.getState()
        << "\n D(x) = "
        << segD.getState()
        << "\n F(x) = "
        << segF.getState()
        << endl << endl;
    cout << "looping with eval()\n";
    for (inputX = 0; inputX < 10; inputX++) {
        segA.eval(inputX);
        segB.eval(inputX);
        segC.eval(inputX);
        segD.eval(inputX);
        segF.eval(inputX);
        cout
            << "Input: " << inputX
            << "\n A(x) = "
            << segA.getState()
            << "\n B(x) = "
            << segB.getState()
            << "\n C(x) = "
            << segC.getState()
            << "\n D(x) = "
            << segD.getState()
            << "\n F(x) = "
            << segF.getState()
            << endl << endl;
    }
    segA.eval(inputX);
    SevenSegmentLogic my7Seg;
    SevenSegmentLogic myCopy(my7Seg);
    for (int inputX = 0; inputX < 16; inputX++) {
        myCopy.eval(inputX);
        cout << "\n| ";
        for (int k = 0; k < 7; k++)
            cout << myCopy.getValOfSeg(k) << " | ";
        cout << endl;
    }
}

BooleanFunc::BooleanFunc(int tableSize, bool evalReturnIfError)
{
    truthTable = new bool[tableSize];
    this->evalReturnIfError = evalReturnIfError;
    this->tableSize = tableSize;
}
BooleanFunc::BooleanFunc(const BooleanFunc& t)
```

```
{
   if (this != &t)
   {
      evalReturnIfError = t.evalReturnIfError;
      tableSize = t.tableSize;
      truthTable = new bool[tableSize];
      for (int i = 0; i < tableSize; i++)
      {
         truthTable[i] = t.truthTable[i];
      }
   }
}
BooleanFunc::~BooleanFunc()
{
   delete[] truthTable;
}
bool BooleanFunc::setTruthTableUsingTrue(int inputsThatProduceTrue[], int arrayS
ize)
{
   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 arra
ySize)
{
   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 (input >= 0 && input < tableSize)
   {
      state = truthTable[input];
```

Repeated code - 1

Null check?

```
      return truthTable[input];
   }
   else
   {
      state = evalReturnIfError;
      return evalReturnIfError;
   }
}
BooleanFunc& BooleanFunc::operator =(const BooleanFunc& t)
{
   if (this != &t)
   {
      evalReturnIfError = t.evalReturnIfError;
      tableSize = t.tableSize;
      delete[] truthTable;
      truthTable = new bool[tableSize];
      for (int i = 0; i < tableSize; i++)
      {
         truthTable[i] = t.truthTable[i];
      }
   }
   return *this;
}

MultiSegmentLogic::MultiSegmentLogic(int numSegs)
{
   segs = new BooleanFunc[numSegs];
   this->numSegs = numSegs;
}
MultiSegmentLogic::~MultiSegmentLogic()
{
   delete[] segs;
}
MultiSegmentLogic::MultiSegmentLogic(const MultiSegmentLogic& t)
{
   if (this != &t)
   {
      delete[] segs;
      numSegs = t.numSegs;
      segs = new BooleanFunc[numSegs];
      for (int i = 0; i < numSegs; i++)
      {
         segs[i] = t.segs[i];
      }
   }
}
bool MultiSegmentLogic::setNumSegs(int numSegs)
{
   if (numSegs < 0)
      return false;
    delete[] segs;
    segs = new BooleanFunc[numSegs];
    this->numSegs = numSegs;
    for (int i = 0; i < numSegs; i++)
    {
       segs[i] = BooleanFunc();
    }
   return true;
}
bool MultiSegmentLogic::setSegment(int segNum, BooleanFunc &funcForThisSeg)
{
   if (segNum < 0 || segNum >= numSegs)
```

reuse in copy ctr.

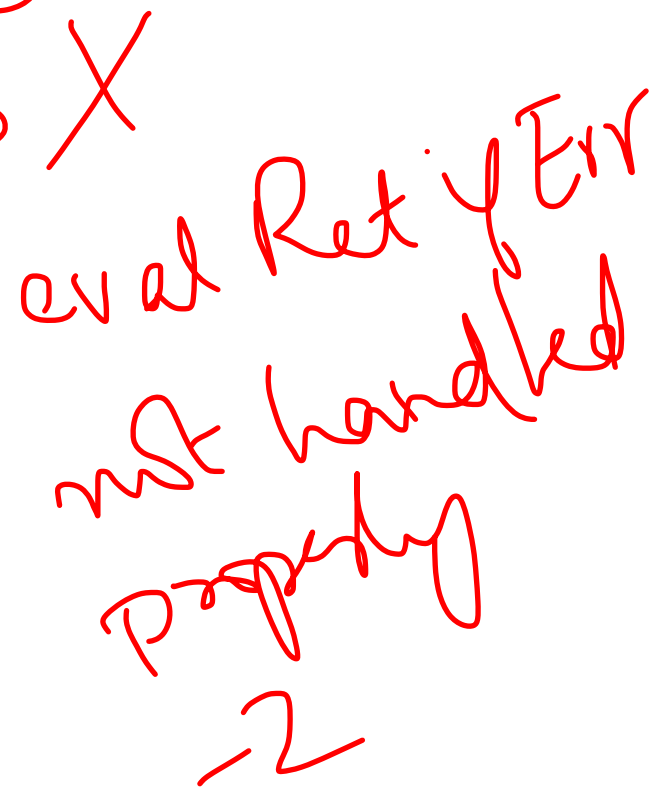
use =op.

```
      return false;

   segs[segNum] = funcForThisSeg;
   return true;
}
void MultiSegmentLogic::eval(int input)
{
   for (int i = 0; i < numSegs; i++)
   {
      segs[i].eval(input);
   }
}

MultiSegmentLogic& MultiSegmentLogic::operator =(const MultiSegmentLogic& t)
{

   if (this != &t)
   {
      delete[] segs;
      numSegs = t.numSegs;
      segs = new BooleanFunc[numSegs];
      for (int i = 0; i < numSegs; i++)
      {
         segs[i] = t.segs[i];
      }
   }
   return *this;
}

SevenSegmentLogic::SevenSegmentLogic()
{
   segs = new BooleanFunc[7];
   this->numSegs = 7;
   init();
}

bool SevenSegmentLogic::getValOfSeg(int seg)
{
   if (seg >= 7 || seg < 0)
   {
      return false;
   }
   else
   {
      return segs[seg].getState();
   }
}
bool SevenSegmentLogic::setSegment(int k, const BooleanFunc& bFunc)
{
   if (k > 6 || k < 0)
   {
      return false;
   }
   else
   {
      segs[k] = bFunc;
      return true;
   }
}
void SevenSegmentLogic::init()
{
   int segmentA[] = {1, 4, 11, 13};
```

X Chain to MSL - 2

Be consistent

```
   BooleanFunc segA = BooleanFunc(16, false);
   segA.setTruthTableUsingFalse(segmentA, 4);
   setSegment(0, segA);

   int segmentB[] = { 5, 6, 11, 12, 14, 15 };
   BooleanFunc segB = BooleanFunc(16, false);
   segB.setTruthTableUsingFalse(segmentB, 6);
   setSegment(1, segB);

   int segmentC[] = { 2, 12, 14, 15 };
   BooleanFunc segC = BooleanFunc(16, false);
   segC.setTruthTableUsingFalse(segmentC, 4);
   setSegment(2, segC);

   int segmentD[] = { 1, 4, 7, 9, 10, 15 };
   BooleanFunc segD = BooleanFunc(16, false);
   segD.setTruthTableUsingFalse(segmentD, 6);
   setSegment(3, segD);

   int segmentE[] = { 1, 3, 4, 5, 7, 9 };
   BooleanFunc segE = BooleanFunc(16, false);
   segE.setTruthTableUsingFalse(segmentE, 6);
   setSegment(4, segE);

   int segmentF[] = { 1, 2, 3, 7, 13 };
   BooleanFunc segF = BooleanFunc(16, false);
   segF.setTruthTableUsingFalse(segmentF, 5);
   setSegment(5, segF);

   int segmentG[] = { 0, 1, 7, 12 };
   BooleanFunc segG = BooleanFunc(16, false);
   segG.setTruthTableUsingFalse(segmentG, 4);
   setSegment(6, segG);
}

/*------------------------------ Posted Run ------------------------------------
-
before eval()

A(x) = 0
B(x) = 0
C(x) = 0
D(x) = 0
F(x) = 0

looping with eval()
Input: 0
A(x) = 1
B(x) = 0
C(x) = 0
D(x) = 1
F(x) = 0

Input: 1
A(x) = 0
B(x) = 0
C(x) = 0
D(x) = 0
F(x) = 0

Input: 2
A(x) = 1
```



B(x) = 0
C(x) = 0
D(x) = 1
F(x) = 0

Input: 3
A(x) = 0
B(x) = 0
C(x) = 0
D(x) = 0
F(x) = 0

Input: 4
A(x) = 1
B(x) = 0
C(x) = 1
D(x) = 1
F(x) = 1

Input: 5
A(x) = 0
B(x) = 0
C(x) = 1
D(x) = 0
F(x) = 1

Input: 6
A(x) = 1
B(x) = 0
C(x) = 1
D(x) = 1
F(x) = 1

Input: 7
A(x) = 0
B(x) = 0
C(x) = 1
D(x) = 0
F(x) = 1

Input: 8
A(x) = 1
B(x) = 0
C(x) = 1
D(x) = 1
F(x) = 1

Input: 9
A(x) = 0
B(x) = 0
C(x) = 1
D(x) = 0
F(x) = 1

| 1 | 1 | 1 | 1 | 1 | 1 | 0 |

| 0 | 1 | 1 | 0 | 0 | 0 | 0 |

| 1 | 1 | 0 | 1 | 1 | 0 | 1 |

| 1 | 1 | 1 | 1 | 0 | 0 | 1 |

```
| 0 | 1 | 1 | 0 | 0 | 1 | 1 |

| 1 | 0 | 1 | 1 | 0 | 1 | 1 |

| 1 | 0 | 1 | 1 | 1 | 1 | 1 |

| 1 | 1 | 1 | 0 | 0 | 0 | 0 |

| 1 | 1 | 1 | 1 | 1 | 1 | 1 |

| 1 | 1 | 1 | 0 | 0 | 1 | 1 |

| 1 | 1 | 1 | 0 | 1 | 1 | 1 |

| 0 | 0 | 1 | 1 | 1 | 1 | 1 |

| 1 | 0 | 0 | 1 | 1 | 1 | 0 |

| 0 | 1 | 1 | 1 | 1 | 0 | 1 |

| 1 | 0 | 0 | 1 | 1 | 1 | 1 |

| 1 | 0 | 0 | 0 | 1 | 1 | 1 |
Press any key to continue . . .
--------------------------------------------------------------------------------
-*/
```

```
// CS 2B Lab 6
// Instructor Solution:
//  Original - Prof. Loceff, Updates, Edits, Annotations:&

// Notes:
//  - Use of sensible names for vars
//  - Correct Boolean logic
//  - Faithfulness to spec
//  - ERROR pattern correctly set (Segments B and C have evalRetIfErr = false)
//  - Correct method qualifications (including virtuals)

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

using namespace std;

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();
};

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();
};

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

private:
    void loadAllFuncs();
};

// ----- BooleanFunc method definitions -----

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 -----

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 -----
```

```
// 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);
}

// ----------------------------------------------------------------------------

// ----- Test driver -----
int main()
{
    BooleanFunc segA, segB(13), segC(100, true);

    // Note: It's good practice to not use sizeof(array[0] in the denom. You
    // don't know if the array might be empty (not in this case, of course)
    int evenFunc[] = { 0, 2, 4, 6, 8, 10, 12, 14 };
    short sizeEvenFunc = sizeof(evenFunc) / sizeof(int);

    int greater9Func[] = { 10, 11, 12, 13, 14, 15 };
    short sizeGreater9Func = sizeof(greater9Func) / sizeof(int);

    int greater3Func[] = { 0, 1, 2, 3 };
    short sizeGreater3Func = sizeof(greater3Func) / sizeof(int);

    segA.setTruthTableUsingTrue(evenFunc, sizeEvenFunc);
    segB.setTruthTableUsingTrue(greater9Func, sizeGreater9Func);
    segC.setTruthTableUsingFalse(greater3Func, sizeGreater3Func);

    // testing class BooleanFunc
    cout << "before eval()\n";
    cout << "\n  A(x) = "
         << segA.getState()
         << "\n  B(x) = "
         << segB.getState()
         << "\n  C(x) = "
         << segC.getState()
         << endl << endl;

    cout << "looping with eval()\n";
    for (int inputX = 0; inputX < 10; inputX++) {
        segA.eval(inputX);
        segB.eval(inputX);
        segC.eval(inputX);
        cout
        << "Input: " << inputX
        << "\n  A(x) = "
        << segA.getState()
        << "\n  B(x) = "
        << segB.getState()
        << "\n  C(x) = "
        << segC.getState()
```

```
        << endl << endl;
    }

    SevenSegmentLogic my7Seg;
    SevenSegmentLogic my7Seg2;
    SevenSegmentLogic myCopy(my7Seg);
    for (int inputX = 0; inputX < 16; inputX++) {
        my7Seg2.eval(inputX);
        cout << std::hex << std::uppercase << inputX << " = | ";
        for (int k = 0; k < 7; k++)
            cout << my7Seg2.getValOfSeg(k) << " | ";
        cout << endl;
    }

    return 0;
}

/* ----- Test run - BooleanFunc -----

before eval()

  A(x) = 0
  B(x) = 0
  C(x) = 1

looping with eval()
Input: 0
  A(x) = 1
  B(x) = 0
  C(x) = 0

Input: 1
  A(x) = 0
  B(x) = 0
  C(x) = 0

Input: 2
  A(x) = 1
  B(x) = 0
  C(x) = 0

Input: 3
  A(x) = 0
  B(x) = 0
  C(x) = 0

Input: 4
  A(x) = 1
  B(x) = 0
  C(x) = 1

Input: 5
  A(x) = 0
```

```
  B(x) = 0
  C(x) = 1

Input: 6
  A(x) = 1
  B(x) = 0
  C(x) = 1

Input: 7
  A(x) = 0
  B(x) = 0
  C(x) = 1

Input: 8
  A(x) = 1
  B(x) = 0
  C(x) = 1

Input: 9
  A(x) = 0
  B(x) = 0
  C(x) = 1

0 = | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
1 = | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
2 = | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
3 = | 1 | 1 | 1 | 1 | 0 | 0 | 1 |
4 = | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
5 = | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
6 = | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
7 = | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
8 = | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
9 = | 1 | 1 | 1 | 1 | 0 | 1 | 1 |
A = | 1 | 1 | 1 | 0 | 1 | 1 | 1 |
B = | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
C = | 1 | 0 | 0 | 1 | 1 | 1 | 0 |
D = | 0 | 1 | 1 | 1 | 1 | 0 | 1 |
E = | 1 | 0 | 0 | 1 | 1 | 1 | 1 |
F = | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
Program ended with exit code: 0
 ------------------------------------------------------------------ */
```