1 OTHER C++ FEATURES

CLASS RandomInt

In Chapter 4, we used a class `RandomInt`, which can be used to conveniently generate random numbers. This section of Appendix 1 presents the source code for this class.

FIGURE A1.1 CLASS RandomInt DOCUMENTATION FILE.

```
/* RandomInt.doc documents RandomInt, a pseudo-random integer class.
 *
 * Written by: Joel C. Adams, Spring, 1993, at Calvin College.
 * Written for: C++: An Introduction To Computing.
 * Revised: Spring 1997, for C++: An Introduction To Computing 2e.
 *
 * Usage: to generate a...
 *  nondeterministic sequence of 'random' numbers in 0..RAND_MAX:
 *     RandomInt rint1;
 *  deterministic sequence of 'random' numbers in 0..RAND_MAX:
 *     RandomInt rint2(a);
 *  nondeterministic sequence of 'random' numbers in b..c:
 *     RandomInt rint3(b, c);
 *  deterministic sequence of 'random' numbers in b..c:
 *     RandomInt rint2(a, b, c);
 *
 * Caution: RandomInt objects are NOT autonomous, but SHARE the ANSI-C
 *  standard random number generator rand(). For most applications,
 *  this will have no effect. However, if a program defines an unseeded
 *  RandomInt r1, and later defines a seeded RandomInt r2, then the
 *  number-sequence for r1 will be 'random' prior to the definition
 *  of r2, but deterministic following the definition of r2 (because r2's
 *  re-seeding the random number generator affects the numbers of r1).
 *
 * Class Invariant: myRandomValue contains a 'random' number
 *                  between myLowerBound and myUpperBound (inclusive).
 *******************************************************************/

// ***** RandomInt.h **************************************************

#ifndef RANDOMINT
#define RANDOMINT
```

```
#include <iostream.h>                              // ostream
#include <stdlib.h>                                // srand(), rand(), RAND_MAX
#include <assert.h>                                // assert()
#include <time.h>                                  // time()

class RandomInt
{
 public:                                           // INTERFACE:
                                                   //  constructors
  RandomInt();                                     //    default values
  RandomInt(int low, int high);                    //    bounded
  RandomInt(int seedValue);                        //    seeded
  RandomInt(int seedValue, int low, int high);     //    seeded & bounded
                                                   //  output:
  void Print(ostream & out) const;                 //    member (<< below)
                                                   //  generate next:
  RandomInt Generate();                            //    default bounds
  RandomInt Generate(int low, int high);           //    new bounds
                                                   //  type-conversion
  operator int();                                  //    RandomInt-to-int

 private:                                          // PRIVATE SECTION:
                                                   //  utility functions
  void Initialize(int low, int high);              //   unseeded initializer
  void SeededInitialize(int seedValue,             //   seeded initializer
                       int low, int high);
  int NextRandomInt();                             //   formula encapsulation

  int       myLowerBound,                          // the minimum random value
            myUpperBound,                          // the maximum random value
            myRandomValue;                         // the current random value

  static bool generatorInitialized;                // initialization flag
};


/*********************************************************************
 * Initialize a RandomInt to a number within default bounds.         *
 * Postcondition: myLowerBound == 0 && myUpperBound == RAND_MAX &&   *
 *                srand() has been seeded (using time()) &&          *
 *                MyRandomValue contains a ‘random’ number.          *
 *********************************************************************/
inline RandomInt::RandomInt()
{
  Initialize(0, RAND_MAX);
}

/*********************************************************************
 * Initialize a RandomInt to a number within specified bounds.       *
 * Precondition: 0 <= low && low < high.                             *
 * Postcondition: myLowerBound == low && myUpperBound == high &&     *
 *                srand() has been seeded (using time()) &&          *
 *                MyRandomValue contains a ‘random’ number.          *
 *********************************************************************/
```

```
inline RandomInt::RandomInt(int low, int high)
{
  assert(0 <= low);
  assert(low < high);
  Initialize(low, high);
}

/*********************************************************************
 * Initialize a RandomInt to a number within default bounds,         *
 *     using a specified seed value.                                 *
 * Postcondition: myLowerBound == 0 && myUpperBound == RAND_MAX &&   *
 *                srand() has been seeded (using seedValue) &&       *
 *                MyRandomValue contains a 'random' number.          *
 *********************************************************************/
inline RandomInt::RandomInt(int seedValue)
{
  assert(seedValue >= 0);
  SeededInitialize(seedValue, 0, RAND_MAX);
}

/*********************************************************************
 * Initialize a RandomInt to a number within specified bounds        *
 *     using a specified seed value.                                 *
 * Precondition: 0 <= low && low < high.                             *
 * Postcondition: myLowerBound == low && myUpperBound == high &&     *
 *                srand() has been seeded (using seedValue) &&       *
 *                MyRandomValue contains a 'random' number.          *
 *********************************************************************/
inline RandomInt::RandomInt(int seedValue, int low, int high)
{
  assert(seedValue >= 0);
  assert(0 <= low);
  assert(low < high);
  SeededInitialize(seedValue, low, high);
}

/**********************************************************************
 * Output a RandomInt via an ostream (Function Member).               *
 * Precondition: out is an ostream.                                   *
 * Postcondition: myRandomValue has been inserted into out.           *
 **********************************************************************/
inline void RandomInt::Print(ostream & out) const
{
  out << myRandomValue;
}

/**********************************************************************
 * Output a RandomInt via an ostream (Non-Function Member).           *
 * Precondition: out is an ostream,                                   *
 *               randInt is a RandomInt.                              *
 * Postcondition: randInt.myRandomValue has been inserted into out && *
 *                ReturnValue == out.                                 *
 **********************************************************************/
```

```
inline ostream & operator<< (ostream & out, const RandomInt & randInt)
{
  randInt.Print(out);
  return out;
}

/*********************************************************************
 * Generate next random integer.                                     *
 * Precondition: srand() has been called to seed rand() &&           *
 *               myLowerBound has been initialized &&                *
 *               myUpperBound has been initialized.                  *
 * Postcondition: ReturnVal == a new 'random' number.                *
 *********************************************************************/
inline int RandomInt::NextRandomInt()
{
  return myLowerBound + rand() % (myUpperBound - myLowerBound + 1);
}

/**********************************************************************
 * Generate a new 'random' number within myLowerBound..myUpperBound.  *
 * Postcondition: myRandomValue has a new 'random' value &&           *
 *               ReturnValue == myself.                               *
 **********************************************************************/
inline RandomInt RandomInt::Generate()
{
  myRandomValue = NextRandomInt();
  return *this;
}

/**********************************************************************
 * Convert RandomInt objects to int objects, where appropriate.       *
 * Precondition: A RandomInt has been used in a context defined       *
 *                for an int, but not for a RandomInt.                *
 * Postcondition: myRandomValue has been read and returned.           *
 **********************************************************************/
inline RandomInt::operator int()
{
  return myRandomValue;
}


#endif


//******* RandomInt.cpp *******************************************

bool RandomInt::generatorInitialized = false;  // initialize the flag

/*********************************************************************
 * Initialize the data members and seed the random number generator  *
 *    using a default value (the clock).                             *
 * Precondition: 0 <= low && low < high.                             *
 * Postcondition: myLowerBound == low && myUpperBound == high &&     *
 *                 srand() has been seeded (using time()) &&         *
 *                 MyRandomValue contains a ‘random’ number.         *
 *******************************************************************/
void RandomInt::Initialize(int low, int high)
{
  myLowerBound = low;
  myUpperBound = high;


  if (!generatorInitialized)       // call srand() first time only
  {
     srand(long(time(0)));        // use clock for seed
     generatorInitialized = true;
  }

  myRandomValue = NextRandomInt();
}

/*******************************************************************
 * Initialize the data members and seed the random number generator  *
 *     using a specified value (seedValue).                          *
 * Precondition: 0 <= low && low < high.                             *
 * Postcondition: myLowerBound == low && myUpperBound == high &&     *
 *                 srand() has been seeded (using seedVal) &&        *
 *                 MyRandomValue contains a ‘random’ number.         *
 *******************************************************************/
void RandomInt::SeededInitialize(int seedValue, int low, int high)
{
  myLowerBound = low;
  myUpperBound = high;
  srand(seedValue);
  generatorInitialized = true;
  myRandomValue = NextRandomInt();
}

/********************************************************************
 * Generate a new 'random' number within low..high.                   *
 * Precondition: 0 <= low && low < high.                              *
 * Postcondition: myRandomValue has a new 'random' value &&           *
 *               ReturnValue == myself.                               *
 ********************************************************************/
RandomInt RandomInt::Generate(int low, int high)
{
  assert(0 <= low);
  assert(low < high);
  myLowerBound = low;
  myUpperBound = high;
  myRandomValue = NextRandomInt();
  return *this;
}
```

FIGURE A1.2 CLASS `RandomInt` HEADER FILE.

```
/* RandomInt.h declares RandomInt, pseudo-random integer class.
 *
 * Written by: Joel C. Adams, Spring, 1993, at Calvin College.
 * Written for: C++: An Introduction To Computing.
 * Revised: Spring 1997, for C++: An Introduction To Computing 2e.
 *
 * See RandomInt.doc for class documentation,
 *     RandomInt.cpp for non-trivial operation definitions.
 *****************************************************************/

#ifndef RANDOMINT
#define RANDOMINT

#include <iostream.h>
#include <stdlib.h>
#include <assert.h>
#include <time.h>

class RandomInt
{
 public:
  RandomInt();
  RandomInt(int low, int high);
  RandomInt(int seedValue);
  RandomInt(int seedValue, int low, int high);
  void Print(ostream & out) const;
  RandomInt Generate();
  RandomInt Generate(int low, int high);
  operator int();

 private:
  void Initialize(int low, int high);
  void SeededInitialize(int seedValue, int low, int high);
  int NextRandomInt();

  int myLowerBound,
      myUpperBound,
      myRandomValue;

  static bool generatorInitialized;
};

inline RandomInt::RandomInt()
{
  Initialize(0, RAND_MAX);
}
```

```
inline RandomInt::RandomInt(int low, int high)
{
  assert(0 <= low);
  assert(low < high);
  Initialize(low, high);
}

inline RandomInt::RandomInt(int seedValue)
{
  assert(seedValue >= 0);
  SeededInitialize(seedValue, 0, RAND_MAX);
}

inline RandomInt::RandomInt(int seedValue, int low, int high)
{
  assert(seedValue >= 0);
  assert(0 <= low);
  assert(low < high);
  SeededInitialize(seedValue, low, high);
}

inline void RandomInt::Print(ostream & out) const
{
  out << myRandomValue;
}

inline ostream & operator<< (ostream & out, const RandomInt & randInt)
{
  randInt.Print(out);
  return out;
}

inline int RandomInt::NextRandomInt()
{
  return myLowerBound + rand() % (myUpperBound - myLowerBound + 1);
}

inline RandomInt RandomInt::Generate()
{
  myRandomValue = NextRandomInt();
  return *this;
}

inline RandomInt::operator int()
{
  return myRandomValue;
}

#endif
```

FIGURE A1.3 CLASS `RandomInt` IMPLEMENTATION FILE.

```
/* RandomInt.cpp defines the non-trivial RandomInt operations.
 *
 * Written by: Joel C. Adams, Spring, 1993, at Calvin College.
 * Written for: C++: An Introduction To Computing.
 * Revised: Spring 1997, for C++: An Introduction To Computing 2e.
 *
 * See RandomInt.h for the class declaration,
 *     RandomInt.doc for class documentation.
 ******************************************************************/

#include "RandomInt.h"

bool RandomInt::generatorInitialized = false;


void RandomInt::Initialize(int low, int high)
{
  myLowerBound = low;
  myUpperBound = high;

  if (!generatorInitialized)       // call srand() first time only
  {
     srand(long(time(0)));        // use clock for seed
     generatorInitialized = true;
  }

  myRandomValue = NextRandomInt();
}


void RandomInt::SeededInitialize(int seedValue, int low, int high)
{
  myLowerBound = low;
  myUpperBound = high;
  srand(seedValue);
  generatorInitialized = true;
  myRandomValue = NextRandomInt();
}


RandomInt RandomInt::Generate(int low, int high)
{
  assert(0 <= low);
  assert(low < high);
  myLowerBound = low;
  myUpperBound = high;
  myRandomValue = NextRandomInt();
  return *this;
}
```

GAUSSIAN ELIMINATION

Section 12.4 illustrated the method of Gaussian Elimination for solving a system of linear equations. The following algorithm summarizes this method. To minimize the effect of roundoff error in the computations, it rearranges the rows to obtain as a pivot that element which is largest in absolute value. Note that if it is not possible to find a nonzero pivot element at some stage, then the linear system is said to be a **singular system** and does not have a unique solution.

GAUSSIAN ELIMINATION ALGORITHM

```
/*  This algorithm solves the linear system $Ax = b$, using Gaussian elimination,
    where $x$ is the $n \times 1$ vector of unknowns.

    Input:    An $n \times n$ coefficient matrix $A$ and an $n \times 1$ constant vector $b$.
    Return:   The solution vector x, if one can be found.
    Output:   A message if the matrix is found to be singular.
-----------------------------------------------------------------*/

 1. a. Input $A$ and $b$.
    b. Form the $n \times (n + 1)$ augmented matrix $Aug$ by adjoining $b$ to $A$:
          $Aug = [A \mid b]$
 2. For $i$ ranging from 1 to $n$, do the following:
    a. Find the entry $Aug[k, i]$, $k = i, i + 1, \ldots, n$ that has the largest absolute value.
    b. Interchange row $i$ and row $k$.
    c. If $Aug[i, i] = 0$, display a message that the linear system is singular and
       stop processing.
    d. For $j$ ranging from $i + 1$ to $n$, do the following:
          Add $\frac{-Aug[j,i]}{Aug[i,i]}$ times the $i$th row of $Aug$ to the $i$th row of $Aug$ to
          eliminate $x[i]$ from the $j$th equation.
 3. a. Set $x[n]$ equal to $\frac{Aug[n,n + 1]}{Aug[n,n]}$.
    b. For $j$ ranging from $n - 1$ to 1 in steps of $-1$, do the following:
          Substitute the values of $x[j + 1], \ldots, x[n]$ in the $j$th equation and
          solve for $x[j]$.
```

The function `GaussElim()` in Figure A1.4 implements this algorithm for Gaussian elimination. Since each of the three steps of the algorithm is nontrival, we have implemented them as the utility functions `ReadEquations()`, `Reduce()`, and `Solve()`.

FIGURE A1.4 GAUSSIAN ELIMINATION.

```
/* GaussElim() performs the Gaussian Elimination algorithm
 *
 * Input: the coefficients of a linear system and its constant vector
 * Return: the solution to the linear system
 * Joel Adams, Fall 1996 at Calvin College.
 *********************************************************************/

#include <iostream.h>     // cout, cin, ...
#include <stdlib.h>       // exit()
#include “Matrix.h”       // Matrix class

void ReadEquations(Matrix & augMat);
int Reduce(Matrix & augMat);
Matrix Solve(Matrix & augMat);

Matrix GaussElim()
{
   Matrix augmentedMatrix;

   ReadEquations(augmentedMatrix);

   bool isSingular = Reduce(augmentedMatrix);

   if (isSingular)
   {
      cerr << "\n*** GaussElim: Coefficient Matrix is (nearly) singular!\n";
      exit (0);
   }

   Matrix solutionVector = Solve(augmentedMatrix);

   return solutionVector;
}


void ReadEquations(Matrix & augMat)
{
  int numEquations;

  for (;;)
    {
      cout << "\nPlease enter the number of equations in the system: ";
      cin >> numEquations;
      if (numEquations > 1) break;
      cerr << "\n*** At least two equations are needed ...\n";
    }

  augMat = Matrix(numEquations,       // numEquations rows
                  numEquations+1);    // numEquations+1 columns
  cout << "\nPlease enter the coefficient matrix row-by-row...\n";
  for (int r = 0; r < numEquations; r++)
    for (int c = 0; c < numEquations; c++)
      cin >> augMat[r][c];

  cout << "\nPlease enter the constant vector...\n";
  for (int r = 0; r < numEquations; r++)
    cin >> augMat[r][numEquations];
}


inline double Abs(double val)
{
  return (val < 0) ? -(val) : val;
}


inline void Swap(double & a, double & b)
{
  double t = a; a = b; b = t;
}

int Reduce(Matrix & augMat)
{
  const double EPSILON = 1.0E-6;
  bool isSingular = false;
  int i = 0,
      j,
      k,
      numRows = augMat.Rows(),
      pivotRow;
  double quotient,
         absolutePivot;

  while ((!isSingular) && (i < numRows))
  {
    absolutePivot = Abs(augMat[i][i]);
    pivotRow = i;
    for (k = i+1; k < numRows; k++)
      if (Abs(augMat[k][i]) > absolutePivot)
      {
         absolutePivot = Abs(augMat[k][i]);
         pivotRow = k;
      }
    isSingular = absolutePivot < EPSILON;
    if (!isSingular)
    {
       if (i != pivotRow)
         for (j = 0; j <= numRows; j++)
           Swap(augMat[i][j], augMat[pivotRow][j]);
       for (j = i+1; j < numRows; j++)
       {
          quotient = -augMat[j][i] / augMat[i][i];
          for (k = i; k <= numRows; k++)
            augMat[j][k] = augMat[j][k] + quotient * augMat[i][k];
        }
    }
    i++;
  }
  return isSingular;
}

Matrix Solve(Matrix & augMat)
{
  Matrix solutionVector(1, augMat.Rows());
  int n = augMat.Rows()-1;

  solutionVector[0][n] = augMat[n][n+1] / augMat[n][n];


  for (int i = n-1; i >= 0; i--)
  {
    solutionVector[0][i] = augMat[i][n+1];

    for (int j = i+1; j <= n; j++)
      solutionVector[0][i] -= augMat[i][j] * solutionVector[0][j];

    solutionVector[0][i] /= augMat[i][i];
  }

  return solutionVector;
}
```

Note that because real numbers are not stored exactly, the statement implementing step 2c checks if the absolute value of element `[i][i]` is less than some small positive number `EPSILON` rather than exactly 0.

Given function `GaussElim()`, we can construct a relatively simple driver program to solve an arbitrary system of linear equations. Figure A1.5 presents such a driver program.

FIGURE A1.5 GAUSSIAN ELIMINATION DRIVER PROGRAM.

```
/* gausselim.cpp is a driver program for Gaussian Elimination.
 *
 * Input:  a series of linear equation coefficients
 * Output: the solution of the linear equation or a “singular
 *           system” message
 * Joel Adams, Fall 1996 at Calvin College.
 **************************************************************/
```

```
#include <iostream.h>     // cout, cin, ...
#include <stdlib.h>       // exit()
#include “Matrix.h”       // Matrix class

Matrix GaussElim();

int main()
{
  cout << "This program solves a linear system "
          "using Gaussian Elimination. \n";

  Matrix solutionVector = GaussElim();

  cout << "\nThe solution vector (x1, x2, ...)  for this system is:\n\n"
       << solutionVector << endl;
}

/* Insert functions from Figure A1.4 here. */
```

Sample runs:

```
This program solves a linear system using Gaussian Elimination.

Please enter the number of equations in the system: 3

Please enter the coefficient matrix row-by-row...
5 -1 -2
-1 5 -2
-2 -2 7

Please enter the constant vector...
11 0 0

The solution vector (x1, x2, ...)  for this system is:

2.84167 1.00833 1.1


This program solves a linear system using Gaussian Elimination.

Please enter the number of equations in the system: 4

Please enter the coefficient matrix row-by-row...
4 4 -5 2
3 3 5 -1
2 1 -1 1
-1 1 -1 1

Please enter the constant vector...
7 9 4 1
```

```
The solution vector (x1, x2, ...)  for this system is:

1   1   1   2


This program solves a linear system using Gaussian Elimination.

Please enter the number of equations in the system: 3

Please enter the coefficient matrix row-by-row...
1 1 1
2 3 4
3 4 5

Please enter the constant vector...
1 2 3

*** GaussElim: Coefficient Matrix is (nearly) singular!
```

GRAPHICS

The PART OF THE PICTURE: Computer Graphics section in Chapter 12 described how a graphics window can be thought of as a two-dimensional array of pixels. Using the Turbo C++ environment, we have constructed a class named `CartesianSystem` that provides an assortment of graphics capabilities. We illustrated its use with two programs, one that used the class to draw the graph of a function *f*(*x*) and one that used the class to draw the density plot of a function *f*(*x*,*y*).

As discussed in this special section of Chapter 12, graphics libraries are not standardized, so that a `CartesianSystem` class that is written for the Turbo environment is not directly portable to the MacIntosh or UNIX environments (although class `CartesianSystem` can be converted to use the graphics libraries of either of these environments, instead of that of the Turbo environment). For the sake of clarity, we have marked each call to a Turbo graphics function with the comment:

```
// GFX funct
```

and each use of a Turbo graphics constant with the comment

```
// GFX const
```

Class `CartesianSystem` can be ported from one environment to another by replacing such calls with the equivalent graphics function in the new environment.

The class declaration is given in Figure A1.6.

FIGURE A1.6 CLASS `CartesianSystem`.

```
/* This class models a cartesian coordinate system.

   ... other documentation omitted ...
---------------------------------------------------------------------*/
```

```
#include <graphics.h>                       // the Turbo graphics lib

#ifndef CARTSYS
#define CARTSYS

typedef double (*FunctionOfX)(double);
typedef double (*FunctionOfXandY)(double, double);


class CartesianSystem
{
  protected:

   // *** Constants ***********************************************
   enum { NUMCOLORS = 7 };                   // density plot colors


   // *** Data Members (Set by Constructor) **********************
   long   colorArray[NUMCOLORS],             // array of the colors
          lastRow,                           // screen size variables
          lastCol,                           // (dependent on monitor)
          bottomMargin,                      // space for prompts
          xAxisRow,                          // useful values in
          yAxisCol;                          //  drawing the axes
   double xMin, xMax,                        // x-axis endpoints
          yMin, yMax,                        // y-axis endpoints
          zMin, zMax,                        // z-axis endpoints
          xDelta,                            // change per pixel
          yDelta,                            //  for x, y and z
          zDelta,
          xAxisIncrement,                    // useful values in
          yAxisIncrement;                    //  drawing the axes


          // *** PRIVATE UTILITY FUNCTIONS ***********

    long round(double realVal)               // round off a double value
    {
        return long (realVal + 0.5);
    }

    long yToRow(double y)                    // map y-value to y-pixel
    {
        return lastRow -                     // pixels run top to bottom
               bottomMargin -                // leave space for margin
               round( (y - yMin) / yDelta);  // scale into y-pixel range
    }

    long xToCol(double x)                    // map x-value to x-pixel
    {
        return round( (x - xMin) / xDelta);  // scale into x-pixel range
    }
```

```
public: // *** PUBLIC INTERFACE FUNCTIONS ***********

  /* --- Class Constructor-------------------------------

      Pre:     A CartesianSystem object has been declared.
      Receive: The extrema for the x, y, and z axes.
      Post:    The screen has been initialized to display
                 a cartesian system with the given extrema.
  ----------------------------------------------------------*/

  CartesianSystem(double xLo, double xHi,
                  double yLo, double yHi,
                  double zLo = 0, double zHi = 0);

  /* --- Class Destructor -------------------------------

      Pre: A CartesianSystem object's lifetime is over.
      Post: The screen is returned to its previous status.
  ----------------------------------------------------------*/
  ~CartesianSystem() { closegraph(); }       // GFX funct

  /* --- DisplayMsg puts a message in the bottom margin.

      Receive: msg, a character string to be displayed.
      Output: msg, within the cartesian system margin.
  ----------------------------------------------------------*/
  void DisplayMsg(char* msg);

  /* --- AwaitUser pauses until the user presses <Enter>.

      Input: a single character.
  ----------------------------------------------------------*/
  void AwaitUser()
  {   char ch; cin.get(ch); }

  /* --- DrawAxes displays labeled x and y axes ------------

      Output: Labeled x and y axes, appropriately scaled.
  ----------------------------------------------------------*/
  void DrawAxes();

  /* --- PlotPoint displays a single (x,y) point ------

      Receive: x and y, two double values.
      Output:  the pixel corresponding to (x,y) is "on".
  ----------------------------------------------------------*/
  void PlotPoint(double x, double y,
                 int color = YELLOW)             // GFX const
      {putpixel(xToCol(x), yToRow(y), color);}   // GFX funct
```

```
    /* --- Graph displays the graph of a function f(x) -------

        Receive: f, a (pointer to a) function f(x).
        Output:  the CartesianSystem graph of fF.
    ---------------------------------------------------------*/
    void Graph(functionOfX f, int color = YELLOW); // GFX const

    /* DensityPlot displays a function f(x, y)'s density plot.

        Receive: f, a (pointer to a) function f(x, y).
        Output:  the CartesianSystem density plot of f,
                 looking down the z-axis.
    ---------------------------------------------------------*/
    void DensityPlot(functionOfXandY f);
};

#endif
```

The functions defined within class `CartesianSystem` contain two calls to Turbo graphics functions:

- The class destructor uses the function `closegraph()` to "turn off" graphics mode.
- The `PlotPoint()` function uses the function `putpixel()` to plot a particular (x,y) point.

The `CartesianSystem` class constructor must do all that is necessary to initialize the screen for graphics display, as well as initialize the various data members of the class. Figure A1.7 presents the definition of the class constructor, minus its leading documentation:

FIGURE A1.7 CLASS `CartesianSystem` CLASS CONSTRUCTOR.

```
CartesianSystem::CartesianSystem(double xLo, double xHi,
                                 double yLo, double yHi,
                                 double zLo, double zHi)
{
    EnterGraphicsMode("C:\\TC\\BGI");    // set display for GFX

    xMin = xLo; xMax = xHi;              // initialize members
    yMin = yLo; yMax = yHi;
    zMin = zLo; zMax = zHi;

    lastRow = getmaxy();                 // GFX funct
    lastCol = getmaxx();                 // GFX funct
    bottomMargin = 15;

    xDelta = (xMax - xMin) / (lastCol+1-bottomMargin);
    yDelta = (yMax - yMin) / (lastRow+1-bottomMargin);
    zDelta = (zMax - zMin) / NUMCOLORS;
```

```
      xAxisIncrement = (xMax - xMin) / 12.0;
      yAxisIncrement = (yMax - yMin) / 10.0;
      xAxisRow = yToRow(0.0);
      yAxisCol = xToCol(0.0);

      colorArray[0] = BROWN;               // GFX const
      colorArray[1] = DARKGRAY;            // GFX const
      colorArray[2] = BLUE;                // GFX const
      colorArray[3] = CYAN;                // GFX const
      colorArray[4] = LIGHTRED;            // GFX const
      colorArray[5] = YELLOW;              // GFX const
      colorArray[6] = WHITE;               // GFX const
  }
```

The class constructor definition contains two calls to Turbo graphics functions:

- `getmaxx()` returns the index of the last (pixel) column on the screen; and
- `getmaxy()` returns the index of the last (pixel) row on the screen.

Note that *x* values map to *columns* and *y* values map to *rows.* A call to a drawing function such as `putpixel()`:

```
putpixel(x, y);
```

thus takes its arguments in (*column, row*) order; ordering them as (*row, column*) will result in a logical error.

In order for graphics functions to have any effect, the constructor must initalize the screen for graphics output. This is accomplished by the function `EnterGraph-icsMode()`, whose definition is in Figure A1.8:

FIGURE A1.8 ENABLING TURBO GRAPHICS.

```
#include <iostream.h>                      // cerr, <<
#include <stdlib.h>                        // exit()

void EnterGraphicsMode(char GraphicsPath[])
{
    int GraphDriver = DETECT;              // GFX const
    int GraphMode;
                                           // GFX function
    initgraph(&GraphDriver, &GraphMode, GraphicsPath);

    int GraphError = graphresult();        // GFX funct

    if (GraphError != grOk)                // GFX const
    {
        cerr << "\nEnterGraphicsMode Error: "
             << grapherrormsg(GraphError)  // GFX funct
             << "\n\n";
        exit (-1);
    }
}
```

The function uses five names defined in Turbo graphics:

- `initgraph()`, a function that initializes the screen for graphics display;
- `DETECT`, a value that instructs `initgraph()` to sense the kind of monitor being used;
- `graphresult()`, a function that indicates the result of the preceding graphics operation;
- `grOk`, one of the constants returned by `graphresult()`; and
- `grapherrormsg()`, a function that given a `graphresult()` return-value, displays an informative error message.

Note that in Figure A1.7, the call

```
EnterGraphicsMode("C:\\TC\\BGI");
```

is used to pass the path to the graphics library, which is in turn passed to `initgraph()`. *This path must be appropriate for your particular system,* and the double back-slashes are needed, since that path is a character string (see Section 2.2).

The axes in a coordinate system provide a meaningful point of reference for whatever other values are displayed. We thus provide a `DrawAxes()` function that displays the axes appropriate for the particular `CartesianSystem` defined by the user of the class. Its definition is shown in Figure A1.9:

FIGURE A1.9 DRAWING `CartesianSystem` AXES.

```
#include <stdio.h>                    // sprintf()

void CartesianSystem::DrawAxes(void)
{                                     // draw x-axis (GFX function)
    line(0, xAxisRow,                 //  starting at this point
         lastCol, xAxisRow);          //  and ending at this point
    line(yAxisCol, 0,                 // draw y-axis (GFX function)
         yAxisCol, lastRow-bottomMargin);

    char buffer[12];                  // label axes
    const int offset = 5;
    long pixelX, pixelY;
                                      // label y-axis
    settextstyle(                     // GFX funct
                 SMALL_FONT,          // GFX const: the font
                 HORIZ_DIR,           // GFX const: horizontal text
                 3);                  // scaling factor
    settextjustify(                   // GFX funct: justification
                   RIGHT_TEXT,        // GFX const: horizontal just.
                   CENTER_TEXT);      // GFX const: vertical just.
                                      // draw the labels
    for (double y = yMax; y >= yMin; y -= yAxisIncrement)
    {
        if ((y > yAxisIncrement/2.0) ||        // avoid crowding
            (y < (-yAxisIncrement)/2.0))       //  the origin
        {
            pixelY = yToRow(y);
            line(yAxisCol-offset, pixelY,      // GFX funct:
                 yAxisCol+offset, pixelY);     //  draw marker
            sprintf(buffer, “%.3f”, y);        // create label
            outtextxy(yAxisCol-offset, pixelY, // GFX funct: draw
                       buffer);                //  string in buffer
        }
    }
                                      // label x-axis
    settextjustify(                   // GFX funct: justification
                   CENTER_TEXT,       // GFX const: horizontal just.
                   TOP_TEXT);         // GFX const: vertical just.
                                      // draw the labels as before
    for (double x = xMin; x <= xMax; x += xAxisIncrement)
    {
        if ((x > xAxisIncrement/2.0) ||         // avoid crowding
            (x < (-xAxisIncrement)/2.0))        //  the origin
        {
            pixelX = xToCol(x);
            line(pixelX, xAxisRow-offset,       // GFX funct
                 pixelX, xAxisRow+offset);
            sprintf(buffer, "%.3f", x);         // create label
            outtextxy(pixelX, xAxisRow+offset,  // GFX funct
                      buffer);
        }
    }
}
```

DrawAxes() is conceptually quite simple, but its implementation requires a number of graphics functions and constants. The details of these can be found in the documentation of GRAPHICS.LIB, or in the Turbo on-line help facility.

Once the axes are drawn, we are ready to graph a function. Figure A1.10 presents the definition of function Graph():

FIGURE A1.10 GRAPHING A FUNCTION.

```
void CartesianSystem::Graph(functionOfX f, int color)
{
    double  x = xMin,                           // init x
            y;
    long  pixelX,
          pixelY;

    DrawAxes();                                 // draw axes
    for (pixelX = 0; pixelX <= lastCol; pixelX++) // for each x-pixel:
    {
        y = (*f)(x);                              // compute y-value
        if ((y >= yMin) && (y <= yMax))           // if it's viewable
        {
            pixelY = yToRow(y);                   //   display it
            putpixel(pixelX, pixelY, color);      //   GFX funct
        }
        x += xDelta;                              // next x
    }
    DisplayMsg("Press <Enter> to continue...");   // prompt user
    AwaitUser();                                  // await response
}
```

`Graph()` simply goes through each x-pixel and plots the appropriate y-value for that pixel. When graphing is completed, it displays a message in the bottom border using `DisplayMsg()`, whose definition of `DisplayMsg()` is given in Figure A1.11:

FIGURE A1.11 DISPLAYING A MESSAGE.

```
void CartesianSystem::DisplayMsg(char* Msg)
{
    settextjustify(                      // GFX funct: justification
                  CENTER_TEXT,           // GFX const: horizontal just.
                  BOTTOM_TEXT);          // GFX const: vertical just.
    settextstyle(                        // GFX funct:
                DEFAULT_FONT,            // GFX const: the font
                HORIZ_DIR,               // GFX const: text orientation
                1);                      // scaling factor
    outtextxy(                           // GFX funct: draw text
              lastCol/2, lastRow,        // at this point
              msg);}
```

Function `Graph()` then uses `AwaitUser()` to suspend execution until the user wants it to continue. The definition of `AwaitUser()` can be seen in Figure A2.1.

The final member of class `CartesianSystem` is function `DensityPlot()`, whose definition is given in Figure A1.12:

FIGURE A1.12 DRAWING A DENSITY PLOT.

```
void CartesianSystem::DensityPlot(functionOfXandY f)
{
    long pixelX,
         pixelY,
         zColor;
    double x,
           y = yMin,
           z;
```

```
                                           // give user feedback
    DisplayMsg("When drawing is complete, press <Enter> to continue.");

                                           // for each row
    for (pixelX = 0; pixelY < lastRow-bottomMargin; pixelY++)
    {
        x = xMin;                          // for each column
        for (pixelX = 0; pixelX < lastCol; pixelX++)
        {
            z = (*F)(x, y);                // compute F(x,y)

            if (z >= zMax)                 // map z into a color
                zColor = NUMCOLORS-1;      // maximum color index
            else if (z <= zMin)
                zColor = 0;                // minimum color index
            else
                zColor = round( (z-zMin)/zDelta );

            putpixel(                      // GFX funct:
                     pixelX, pixelY,       //  plot a pixel of the
                     colorArray[zColor]);  //  appropriate color

            x += xDelta;                   // next x
        }
        y += yDelta;                       // next y
    }

    DrawAxes();                            // draw axes when done
    AwaitUser();                           // await user response
}
```

The preceding figures demonstrate the use of the `putpixel()`, `line()`, and `outtextxy()` functions that draw points, lines, and text, respectively. In addition to these functions, Turbo's `GRAPHICS.LIB` library provides the functions:

`void circle(int` *`pixelX`*`, int` *`pixelY`*`, int` *`radius`*`)`

that can be used to draw a circle on the screen of radius *`radius`* centered at (*`pixelX`*, *`pixelY`*); and

```
void rectangle(int upperLeftCornerX, int upperLeftCornerY,
               int lowerRightCornerX, int lowerRightCornerY);
```

that can be used to draw rectangles (boxes) on the screen; and the more general

```
void  drawpoly(int numberOfVertices, int vertexArray[]);
```

that can be used to draw an arbitrary polygon, whose vertices'(x, y) coordinate pairs are stored in *`vertexArray`*.

Each of these capabilities can be incorporated into class `CartesianSystem` by adding a member function that calls the appropriate function. For example, a `DrawBox()` function for class `CartesianSystem` can be defined as follows:

```
void DrawBox(int topLeftX, int topLeftY, int bottomRightX, int bottomRightY)
{
   rectangle(xToCol(topLeftX), yToRow(topLeftY),
             xToCol(bottomRightX), yToRow(bottomRightY));
}
```

These primitive graphics operations can then be used to construct more complicated graphics objects, such as bar graphs, pie charts, histograms, windows, menus, and so on.

There are many, many more capabilities provided by Turbo's graphics library (and those of the other environments). For more information, the interested reader is encouraged to explore the Turbo on-line help facility and/or the system documentation.

Default Values for Parameters in Functions

Problem. We wish to construct a function that will evaluate any real-valued polynomial function of degree 4 or less for a given real value x. The general form of such a polynomial is

$$a + bx + cx^2 + dx^3 + ex^4$$

where the coefficients a, b, c, d, and e are real constants.

One Solution. The function in Figure A1.13 is one way to solve the problem. Its specification is:

Receive:	The real values $a, b, c, d, e,$ and x
Return:	The real value $a + bx + cx^2 + dx^3 + ex^4$

It simply returns the value of an appropriate C++ expression for $a + bx + cx^2 + dx^3 + ex^4$.

Figure A1.13 Evaluating Polynomials—Version 1.

```
/* Polynomial will evaluate any polynomial up to degree 4.
 *
 * Receive: the real values x, a, b, c, d, and e
 * Return:  the real value a + bx + cx^2 + dx^3 + ex^4
 ******************************************************************/

#include <math.h>

double Polynomial(double x, double a, double b,
                  double c, double d, double e)
{
   return a + b*x + c*pow(x, 2.0) + d*pow(x, 3.0) + e*pow(x, 4.0);
}
```

The Difficulty. The difficulty with `Polynomial()` is that to make `Polynomial()` a (relatively) general function, we used six parameters. In applications that involve lower-degree polynomials, however, this generality becomes a nuisance, because we must pass arguments for all of the parameters each time we call `Polynomial()`. For example, to calculate the value of

$$P(x) = 4 + 4x + x^2$$

when x is 12, we would use the call

```
yVal = Polynomial(12.0, 4.0, 4.0, 1.0, 0.0, 0.0);
```

and to compute the value of the linear function

$$Q(x) = 3 - 4x$$

when x is 7, we would use the call:

```
yVal = Polynomial(7.0, 3.0, -4.0, 0.0, 0.0, 0.0);
```

The Solution. To avoid this inconvenience, C++ allows the programmer to specify a **default value** for a parameter. If a function with default-valued parameters is called and no arguments are passed to these parameters, they receive their specified default values.

To provide a default value for a parameter, we simply use the assignment operator (`=`) to give the parameter its default value, usually in the prototype of the function. For example, suppose we change the heading of `Polynomial()` to

```
double Polynomial(double x, double a = 0.0, double b = 0.0,
                  double c = 0.0, double d = 0.0, double e = 0.0)
```

Now if we need to compute the value

$$P(x) = 4 + 4x + x^2$$

when x is 12, we can use the call:

```
yVal = Polynomial(12.0, 4.0, 4.0, 1.0);
```

and since only four arguments are passed, the last two parameters (`d` and `e`) are given their default values (0.0). Similarly, to compute the function

$$Q(x) = 3 - 4x$$

when x is 8, we can use the call

```
yVal = Polynomial(8.0, 3.0, -4.0);
```

and since only three arguments are passed, the last three parameters (`c`, `d`, and `e`) will be given their default values.

Limitations in Using Default Parameter Values. There are several restrictions that C++ imposes on the use of parameters with default values:

1. *Default values for parameters of a function can be given only once.* The practical implication of this rule is that *a parameter should be given a default value in the prototype of the function, or in its definition, but not both.*
2. *If the default parameter values are given in the prototype of the function, and*

that prototype is stored in the header file of a library, then any program that uses the `#include` *directive to insert that header file can use the default values.* This is the recommended approach.

3. *If the default values are given in the definition of the function, and the function's definition is stored in the implementation file of a library, then the default values cannot be used by a program that inserts the header file of that library (using the* `#include` *directive)* . This is because a program that uses a library (by inserting its header file) "sees" the prototypes in the header file of the library but never "sees" the function definitions in the implementation file. Consequently, we suggest that the default values be given in the function prototype, since otherwise the default parameter values cannot be used outside the library.
4. *If an argument is supplied for a parameter p that has a default value, then an argument must be supplied for every parameter that precedes p in the parameter list.* For example, suppose that we wish to evaluate

   $$P(x) = 2 \ \ + 3x^2$$

   when x is 1. Then we are unable to take advantage of the default value of parameter `b` and must use the expression:

   ```
   Polynomial(1.0, 2.0, 0.0, 3.0)
   ```

   to call `Polynomial()`. The reason for this is that in determining which argument goes with which parameter, C++ matches from left to right, associating the first argument with the first parameter, the second argument with the second parameter, and so on. Thus, the expression:

   ```
   Polynomial(1.0, 2.0, 3.0)
   ```

   evaluates the polynomial

   $$2 + 3x$$

   instead of the polynomial we intended. Note that making this mistake results in a *logic* error, not a *syntax* error. The compiler will process such calls without generating an error.
5. *Parameters that receive default values must be declared at the end of the parameter list.* Stated differently, a parameter that does not receive a default value must precede all parameters that do. For example, a syntax error results if we try to declare `Polynomial()` as

```
double Polynomial(double x, double a = 0.0, double b,
                  double c = 0.0, double d = 0.0, double e = 0.0);
```

   because parameter `a,` which has a default value, precedes parameter `b`, which has no default value. Again, this rule makes sense, since otherwise a call like

   ```
   Polynomial(3.0, 2.0, 1.0);
   ```

   is ambiguous because it is not clear which default values the programmer intended to use. While `x` is clearly meant to be 3.0, the programmer could have intended that a be 2.0 and b be 1.0; or that a be 0.0, b be 2.0 and c be 1.0; or that a be 0.0, b be 2.0, c be 0.0, and d be 1.0; and so on.

Varying the Number of Arguments in Functions

The default parameter value mechanism allows us to call a function with *fewer* arguments than the specified number of parameters. Now, we consider the problem of constructing a function that can be called with *more* arguments than the number specified. To illustrate, consider the following generalization of the preceding polynomial problem.

Problem. We wish to construct a function that will evaluate a polynomial of degree *n*, for any positive integer *n.*

Solution. A specification for the function is

Receive: the *degree* of the polynomial
x, the value at which the polynomial is to be evaluated
a, the constant term in the polynomial
the coefficients of higher-order terms (if any)

Return: The value of the polynomial at *x*

To compute the value of the polynomial, we can use the following algorithm:

Algorithm for Evaluating a Polynomial

```
1. Initialize polyValue to a, and power_of_x to 1.0.
2. For each value i in the range 1 through degree:
    a. Get the ith coefficient of the polynomial, storing it in nextCoef.
    b. Multiply power_of_x by x.
    c. Multiply nextCoef by powerOfX and add the product to polyValue.
3. Return the value of polyValue.
```

Coding. To code the function, we need a mechanism for *passing different numbers of arguments to the function,* depending on the degree of the polynomial we want to use. We need a function in which there is no limit on the number of arguments. A stub for a function that accomplishes this is as follows:

```
#include <stdarg.h>  // declarations to permit
                     // varying numbers of arguments

double Polynomial(int degree, double x, double a, ...)
{
}
```

The declarations in the stdarg library allow the use of **ellipses** (. . .) within the parameter list to inform the C++ compiler that if this function is called with more than three arguments, the extra parameters should not be treated as errors. The compiler thus "turns off" argument-checking when processing such calls. It is left to programmers that use this ellipses mechanism to ensure that the number and type of arguments are correct.

When a function whose parameter list includes ellipses is called, any additional arguments that are present are placed into a special type of list, called a `va_list`(for varying-argument list). The type `va_list` and the operations for manipulating it are declared in the library `stdarg`, so its header file must be inserted (using the `#include` directive) before the function definition. There are three basic operations for manipulating a varying-argument list. In the following descriptions of these operations, *`list`* is of type `va_list`:

- `va_start(`*`list, lastParam`*`)`: initializes *`list`* for processing; *`lastParam`* is the name of the last parameter in the function declaration.
- `va_arg(`*`list, type`*`)`: retrieves and returns the next value of the specified *`type`* from *`list`* (assuming *`list`* has been initialized with `va_start()`).
- `va_end(`*`list`*`)`: "cleans up" *`list`* after processing is completed.

This type `va_list` and the preceding operations make it possible to implement the polynomial-evaluation algorithm as shown in Figure A1.14.

FIGURE A1.14 EVALUATING POLYNOMIALS—VERSION 2.

```
/* Polynomial will evaluate a polynomial of any degree.
 *
 * Receive: the int degree, a real value x, and the real
 *          coefficients a, ... of a polynomial
 * Return:  the real value of the polynomial at x
 *******************************************************************/

#include <stdarg.h>

double Polynomial(int degree, double x, double a, ...)
{
   double
      power_of_x = 1.0,                 // powers of x
      nextCoef,                         // next coefficient
      polyValue = a;                    // polynomial's value at x

   va_list argList;

   va_start(argList, a);                // argList begins after a

   for (int i = 1; i <= degree; i++)
   {
      power_of_x *= X;                  // i-th power of X
      nextCoef = va_arg(argList, double); // get the ith coefficient

      polyValue += nextCoef * powerOfX;   // ith term of polynomial
   }

   va_end(argList);                     // clean up the list

   return polyValue;
}
```

Figure A1.15 gives a driver program that tests this function for polynomials of degree ≤ 3.

FIGURE A1.15 A DRIVER PROGRAM FOR `Polynomial()`.

```
/* polytester.cpp is a driver program to test function Polynomial().
 *
 * Output: the value of Polynomial() for polynomials of various degrees
 *******************************************************************/

#include <iostream.h>

double Polynomial(int degree, double x, double a, ...);

int main()
{
   cout // P(1.0) for P(x) = 2
        << Polynomial(0, 1.0, 2.0) << endl

        // P(1.0) for P(x) = 2 + 3x
        << Polynomial(1, 1.0, 2.0, 3.0) << endl

        // P(1.0) for P(x) = 2 + 3x + 4x^2
        << Polynomial(2, 1.0, 2.0, 3.0, 4.0) << endl

        // P(1.0) for P(x) = 2 + 3x + 4x^2 + 5x^3
        << Polynomial(3, 1.0, 2.0, 3.0, 4.0, 5.0) << endl;

   return 0;
}

/* Insert the #include directive and the definition
   of function Polynomial() from Figure D.2 here.  */
```

Sample run:

```
2
5
9
14
```

Note that it is the programmer's responsibility to ensure that `Polynomial()` is called correctly. If we were to erroneously pass `int` values instead of `double` values to `Polynomial()`,

```
Polynomial(4, 1, 2, 3, 4, 5)
```

then the arguments would be stored as `int` (instead of `double`) values within the `va_list`. If `int` values are stored in one memory word and `double` values in two memory words, then the first call to `va_arg()`, `va_arg(argList, double)`, would interpret the two (`int`) words storing 3 and 4 as a `double` value, and consequently return an erroneous result. This is one situation where different types of numeric data cannot be freely intermixed: If a function that uses this ellipses mechanism is expecting a series of arguments of a particular type, then it must receive arguments of that type.