The GF2Matrices Classes:
A Programming Package for Mathematical Research

Peter M. Maurer
Dept. of Computer Science
Baylor University
Waco, Texas 76798

1 Abstract
Over the past few years I have been engaged in an intense study of GF(2) matrices, especially of dimensions 2, 3, 4, and 5. The software I used for this study was mostly a bunch of ad-hoc subroutines scattered over numerous projects. This package consolidates all of this software into a system of classes. This software can be used to reproduce any of my results.

2 Introduction
The GFM package contains definitions of GF(2) vectors, matrices and collections for dimensions 2, 3, 4, and 5. The header file GF2Matrices.h contains the definitions of all classes, and the bodies of all type-conversion operator overloads. The file GF2Matrices.cpp contains all function bodies except those for type-conversion operator overloads. The header file will define “NULL” as zero (0) if it is not already defined. The header file contains #defines to prevent double-loading.

Six classes are defined for each dimension. These are Vector, VectorList, Orbit, Matrix, Group, and GroupList. The VectorList is an ordered collection of vectors, while Orbit is an ordered collection of VectorList’s. Similarly, Group is an ordered collection of matrices, while GroupList is an ordered collection of Group’s. For each dimension, the class name is suffixed with the dimension, so for dimension 2, the six classes are Vector2, VectorList2, Orbit2, Matrix2, Group2, and GroupList2. The four collection classes are ordered but not sorted. They may contain duplicates. It is possible to create linked lists of any of the objects. In fact, the collections are maintained by using the linking properties of the contained objects.

In addition to the 24 base classes, there are eight derived classes that are used to enumerate matrices. Each of the matrix classes has two derived classes, GenerateGLG, and GenerateAll. These are suffixed with the dimension so for dimension two, the classes are GenerateGLG2 and GenerateAll2. Most operations on 5x5 matrices are memory intensive, so the enumerator classes are necessary for 32-bit operating systems. The enumerators are provided for the other dimensions as a convenience. The classes GenerateGLG (GLG=General Linear Group) generate all non-singular matrices of a particular size. The technical term for the set of all non-singular matrices of a particular dimension is the General Linear Group of size n. These groups are of the following sizes.

5 by 5 — 9,999,360
4 by 4 — 20,160
The GenerateAll classes generate all matrices, including the singular ones. These sets of matrices are of the following sizes.

- 5 by 5 — 33,554,432
- 4 by 4 — 65,768
- 3 by 3 — 512
- 2 by 2 — 16

The generator classes do not generate and store the entire set of matrices. Instead, functions are provided to enumerate the matrices one-by-one. These functions transform the base matrix from one matrix to the next. In most cases, objects of the generator class can be used wherever objects of the base class can be used.

None of the classes defined in this package has any public data items. The functions are described in the next sections.

3 Class Functions

Each class defines four basic functions: the default constructor, the destructor, the copy constructor and the assignment operator. The remaining functions are divided into the following classes: Constructors, Type Converters, Operator Overloads, Accessors and Utility Functions. Those functions that perform tests return integers, zero for false and one for true. (I hate Booleans.)

3.1 Vector Functions

Most vector functions do not require indexing of vector elements, but some knowledge of the indexing scheme is necessary to get things to work right. The elements of each vector are indexed starting from zero. Conceptually, the indices read from left to right starting from zero, so when a vector is displayed in any fashion, the zero-index element will appear at the left. Indices are used explicitly in the Set and Get functions.

3.1.1 Constructors

In addition to the default and copy constructors, each vector defines a number-list constructor as illustrated below.

```cpp
V = Vector2(0,1);
Vector3 W(1,0,1);
Vector4 P(1,1,0,0);
Q = Vector5(1,0,0,1,0);
```

The operands of these constructors should be specified as ones or zeros, but if other integers are specified, only the low-order-bit of the integer will be used. Thus 0, 2, 4, and 5000 will all initialize an element of a vector to zero.

Vector elements are initialized from low index to high index using the operands from left to right. So the constructor Vector2(0,1) initializes element 0 to 1 and element 1 to zero.
3.1.2 Type Converters

Two different kinds of type converters are provided, constructors and operator overloads. The bodies of the operator overloads are defined within the class definition, while the bodies of the constructors are defined in the GF2Matrices.cpp file. Type conversions are provided from integer to vector and vice versa, integer to string and vice versa, and promotions from lower dimensional vectors. Demotions from a higher dimension to a lower are provided using accessor functions. Type converter constructors can be specified explicitly, but it is not necessary to do so. Thus, the following code is legal.

```cpp
Vector5 V;
V = 7;
int x = V;
```

When initializing a vector with an integer, only the low-order bits will be used. The low order bit of the integer initializes the element with the highest index so that the two vectors illustrated below will be initialized identically. This is done to make the vectors read correctly when viewing them as binary numbers.

```cpp
Vector5 V(0,0,0,0,1), W;
W = 1;
```

The vector-to-string conversions produce a newly allocated string that must be deleted when it is no longer needed. One can print a vector using the following set of statements.

```cpp
Vector4 V;
char *S;
...
S = V;
cout<<S<<endl;
delete [] S;
```

When upgrading from a lower dimension to a higher dimension, the low-index positions of the higher dimensional vector are filled first. (i.e. the mapping is between identical indices.) Thus, the vectors V and W are initialized identically.

```cpp
Vector5 V(1,1,0,0,0), W;
W = Vector2(1,1);
```

3.1.3 Operator Overloads

The assignment operator works as expected, but the NEXT pointer is never altered so assignment will not change the linking status of an object. In addition to the assignment operator, the vector classes provide overloads for the comparison operators and for addition and multiplication operators. (Subtraction is the same as addition in GF(2).)

For the comparison operators, two vectors are equal if the elements at all index positions are equal, and are unequal otherwise. For less-than and greater-than operations, the vectors are compared element-by-element in ascending order by index. The first differing pair (if any) determines the result of the comparison. One is considered larger than zero. All six comparison operators, ==, !==, <, >, <= and >=, are implemented.
Addition is standard element-by-element vector addition. Note that in GF(2) the exclusive-or function is used to perform addition, so Vector4(1,1,0,0)+Vector4(1,0,1,0) will produce Vector4(0,1,1,0). Both the + and += operators are implemented.

There are three multiply operators, multiplication by a vector Vector2(1,1)*Vector2(1,0), which produces the dot-product, multiplication by a constant Vector2(1,1)*1, which produces the vector itself or the zero vector, and multiplication by a matrix Vector2(1,1)*Matrix2(0,1,1,0), which applies the matrix to the vector and produces another vector.

For multiplication by a constant and multiplication by a matrix, both the * and *= operators are implemented. For dot product, only * is implemented. Multiplication by a matrix uses the standard matrix-multiplication algorithm. For dot-product and matrix multiplication, addition is performed using exclusive-or. The += and *= operators return a reference (as does the assignment overload) so things like V1 = V2 *= M; are legal.

When multiplying a vector by an integer, the vector must be on the left, and the integer should be either one or zero. If any other integer is specified, only the low-order bit will be used.

Dot product returns an integer while multiplication by an integer or a matrix returns a vector.

3.1.4 Accessors

Accessor functions, as their name implies, permit one to access various parts of a vector. Although vectors have no public data items, there are sufficient accessor functions to permit controlled access to any part of the vector.

All vectors implement the following functions.

- GetNext( ) – returns the Next pointer. Used for linked lists.
- SetNext(Vectorx * NewNext) – sets the Next pointer. Used for linked lists.
- Get(int x) – return an integer value corresponding to position x. Returns 0 if x is out of range
- Set(int x, int NewVal) – Sets position x to NewVal. Only the low order bit of NewVal is used. No-ops if x is out of range.
- int Order(void) – Returns the number of 1’s in a vector. Another term for Weight.

In addition to these five functions, the classes Vector3, Vector4, and Vector5 implement demotion functions to convert vectors to lower dimensions.

- GetV2(int x) – This function returns a Vector2 starting at position x. A zero vector is returned if the x is not valid or is too large to permit two consecutive elements to be selected from the vector. Implemented by Vector3, Vector4 and Vector5.
- GetV3(int x) – This function returns a Vector3 starting at position x. A zero vector is returned if the x is not valid or is too large to permit three consecutive elements to be selected from the vector. Implemented by Vector4 and Vector5.
- GetV4(int x) – This function returns a Vector4 starting at position x. A zero vector is returned if the x is not valid or is too large to permit four consecutive elements to be selected from the vector. Implemented by Vector5.

3.1.5 Utility Functions.

Each of the four vector classes implements the following functions.
int isZero( ) – tests whether the vector is all zeros.
int isOne( ) – tests whether the vector is all ones.
Serialize( ) – converts the vector to ASCII for printing. This function returns a char * that
must be deleted when no longer needed.
SerializeNL( ) – converts the vector to ASCII for printing, and adds a newline to the end.
    This function returns a char * that must be deleted when no longer
needed.
IntVal( ) – converts the vector to an integer and returns the result. Has the same effect as
assigning the vector to an integer variable.

4 Matrix Functions

The elements of each matrix are indexed by row and column starting with zero in the upper
left. Thus, the upper left element is element 0,0. When matrices are displayed, they are normally
displayed in row/column format with element 0,0 at the upper left. In some cases, all rows of a
matrix are concatenated into a single line, in ascending order by index.

4.1 Constructors and Type Converters

The default constructor creates an identity matrix with ones on the main diagonal and zeros
elsewhere. In addition to the default and copy constructors, there are many other constructors
that allow matrices to be assembled in various ways. The elements of the matrix can be specified
as a list of numbers as in the following examples. These numbers should be specified as either 1
or 0, but if other integers are specified, only the least significant digit will be used.

Matrix2 A(1,0, 0,1);
Matrix3 B(1,0,0, 0,1,0, 0,0,1);
Matrix4 C(1,0,0, 0,1,0,0, 0,0,1,0, 0,0,0,1);
Matrix5 D(1,0,0, 0,0, 0,1,0,0,0, 0,0,1,0,0, 0,0,0,0,1);

A diagonal matrix can be created by specifying only enough values for the main diagonal as
follows. In the following examples, the specified values will be used to initialize the main
diagonal and the remaining elements will be set to zero.

Matrix2 A(1,1);
Matrix3 B(1,1,1);
Matrix4 C(1,1,1,1);
Matrix5 D(1,1,1,1,1);

As before, if anything other than a 1 or 0 is specified, only the low-order bit will be used.
Matrices can be initialized using vectors of the proper size. If a single vector is specified, it
will be used to initialize the main diagonal, and the remaining values will be set to zero.
Otherwise, it is necessary to specify one vector per row as in the following examples.

Matrix2 A(Vector2(2),Vector2(1));
Matrix3 B(Vector3(4), Vector3(2),Vector3(1));
Matrix4 C(Vector4(8),Vector4(4),Vector4(2),Vector4(1));
Matrix5 D(Vector5(16),Vector5(8),Vector5(4),Vector5(2),Vector5(1));
The single-vector constructor can be used as a type converter to convert vectors to matrices. A reverse type conversion is available for converting matrices to vectors, so the following code is legal.

```cpp
Vector4 V;
Matrix4 M;

V = M;
M = V;
```

When a matrix is assigned to a vector (of the proper size) the elements of the main diagonal are copied into the vector.

Matrices can also be constructed from integers and strings. These constructors can act as type converters. Reverse type conversions are available for these constructors, so assignments may be made in either direction.

When a matrix is constructed from an integer, the low order bit of the integer is used to initialize the element with the highest row and column indices. Bits are used from low-order to high-order to initialize the matrix row-by-row from high-indices to low indices. Thus the constructors specified below create the corresponding matrices. An integer value of zero creates the zero matrix.

```cpp
Matrix5 M(1);
(0 0 0 0 0)
(0 0 0 0 0)
(0 0 0 0 0)
(0 0 0 0 0)
(0 0 0 1 0)
```

```cpp
Matrix5 M(2);
(0 0 0 0 0)
(0 0 0 0 0)
(0 0 0 0 0)
(0 0 0 0 0)
(0 0 0 1 0)
```

```cpp
Matrix5 M(32);
(0 0 0 0 0)
(0 0 0 0 0)
(0 0 0 0 0)
(0 0 0 0 1)
(0 0 0 0 0)
```
When a string is used to initialize a matrix, the matrix is first initialized as the identity matrix, and the characters of the string are used to overwrite the entries of the identity matrix. The specified string may have fewer characters than there are matrix elements. Any extra characters will be ignored. The characters of the string should be specified as ‘1’ or ‘0’, but if any other character is specified, only the least significant bit will be used. The string will be processed from left to right, and the resulting values will be assigned to matrix elements in a row-by-row fashion from the low indices to the high indices. Thus, the first character of the string will be used to initialize element (0,0). The string may be empty, but must not be specified as NULL.

Matrices of sizes 3x3, 4x4, and 5x5 can be constructed from smaller matrices. The smaller matrices are placed along the main diagonal of the new matrix so their diagonal elements will coincide. Integer operands are treated as 1x1 matrices. Before using any operands, these constructors will initialize the matrix to all zeros. The specified elements are then used to overwrite the appropriate zero elements. All possible combinations of operands are provided.

The following constructors are available.

4.1.1 3x3 from 2x2
Matrix3(Matrix2,int)
The 2x2 matrix is inserted at the upper left, and the integer is inserted into the lower right element
Matrix3(int,Matrix2)
The 2x2 matrix is inserted at the lower right, and the integer is inserted into the upper left element.

4.1.2 4x4 from 3x3
Matrix4(Matrix3,int)
The 3x3 matrix is inserted at the upper left, and the integer is inserted into the lower right element
Matrix4(int,Matrix3)
The 3x3 matrix is inserted at the lower right, and the integer is inserted into the upper left element.

4.1.3 4x4 from 2x2
Matrix4(Matrix2,int,int)
The 2x2 matrix is inserted at the upper left, the first integer is inserted into element (2,2) and the second integer is inserted into element (3,3)
Matrix4(int,Matrix2,int)
The 2x2 matrix is inserted with its upper left in position (1,1). The first integer is inserted into element (0,0) and the second integer is inserted into element (3,3)
Matrix4(int,int,Matrix2)
The 2x2 matrix is inserted at the lower right. The first integer is inserted into element (0,0) and the second integer is inserted into element (1,1)
Matrix4(Matrix2,Matrix2)
The first matrix is inserted at the upper left and the second is inserted at the lower right.

4.1.4 5x5 from 4x4
Matrix5(Matrix4,int)
The 4x4 matrix is inserted at the upper left, and the integer is inserted into the lower right element.

**Matrix5(int, Matrix4)**
The 4x4 matrix is inserted at the lower right, and the integer is inserted into the upper left element.

**4.1.5 5x5 from a 3x3 and a 2x2**

**Matrix5(Matrix3, Matrix2)**
The 3x3 matrix is inserted at the upper left, and the 2x2 is inserted at the lower right.

**Matrix5(Matrix2, Matrix3)**
The 2x2 matrix is inserted at the upper left, and the 3x3 is inserted at the lower right.

**4.1.6 5x5 from 3x3**

**Matrix5(Matrix3, int, int)**
The 3x3 matrix is inserted at the upper left. The first integer is inserted into element (3,3) and the second is inserted into element (4,4).

**Matrix5(int, Matrix3, int)**
The 3x3 matrix is inserted so its upper left corner is at position (1,1). The first integer is inserted into element (0,0) and the second is inserted into element (4,4).

**Matrix5(int, int, Matrix3)**
The 3x3 matrix is inserted at the lower right. The first integer is inserted into element (0,0) and the second is inserted into element (1,1).

**4.1.7 5x5 from 2x2**

**Matrix5(Matrix2, Matrix2, int)**
The first 2x2 matrix is inserted at the upper left, the second is inserted so its upper left is at position (2,2) and the integer is inserted into position (4,4).

**Matrix5(Matrix2, int, Matrix2)**
The first 2x2 matrix is inserted at the upper left, the second is inserted at the lower right, and the integer is inserted into position (2,2).

**Matrix5(int, Matrix2, Matrix2)**
The first 2x2 matrix is inserted so its upper left corner is at position (1,1), the second is inserted at the lower left and the integer is inserted into position (0,0).

**Matrix5(Matrix2, int, int)**
The 2x2 matrix is inserted at the upper left. The integers are inserted into positions (2,2), (3,3) and (4,4) respectively.

**Matrix5(int, Matrix2, int)**
The 2x2 matrix is inserted so its upper left corner is at position (1,1). The integers are inserted into positions (0,0), (3,3) and (4,4) respectively.

**Matrix5(int, int, Matrix2)**
The 2x2 matrix is inserted so its upper left corner is at position (2,2). The integers are inserted into positions (0,0), (1,1) and (4,4) respectively.

**Matrix5(int, int, int, Matrix2)**
The 2x2 matrix is inserted at the lower right. The integers are inserted into positions (0,0), (1,1) and (2,2) respectively.
4.1.8 Promotion Type Converters

Finally, the classes Matrix3, Matrix4, and Matrix5 provide type-conversion constructors that will promote matrices of any lower dimension to the current type. These constructors first initialize the new matrix to the identity matrix and then copy the argument matrix into the upper left corner.

4.2 Operator Overloads

The assignment overload works as expected, but does not alter the NEXT pointer, so assignment will not alter the linking status of a Matrix object.

In addition to the assignment operator, the Matrix classes overload all six comparison operators, ==, !=, <, <=, >, >=, and the multiply and addition operators *, *+, +, and *+. Two matrices are equal if all corresponding positions have the same value. When matrices are not equal it is the first unequal position determines the order of the matrices. The first differing position is found by scanning rows from left to right starting with the top row, and proceeding row-by-row to the bottom. One is considered larger than zero.

The operators + and += perform standard matrix addition. That is, corresponding elements in the same position are added (using the exclusive or operation), and a matrix containing the results is returned. A matrix can be multiplied by a vector or by another matrix. When the operand is another matrix, either the * operator or the *+ operator can be used. The result is a standard matrix multiplication operation on the two matrices. Exclusive or is used for addition. When the operand is a vector, only the * operator may be used. The result is a vector obtained by treating the vector operand as an nx1 column matrix and performing standard matrix multiplication using exclusive or for addition.

The *+, += and = operators return references to the altered object, so assignment operators may be chained. Self-referential operations like A*+=A; will operate correctly.

4.3 Accessors

Access to the NEXT pointer is provided with the functions GetNext( ) and SetNext( ). GetNext returns a pointer, while SetNext has a single operand which is assigned to the NEXT pointer. The operand of SetNext is not audited in any way. Why not just make the NEXT pointer public? I’m not sure. It seemed like a good idea at the time.

Access to matrix elements is provided through the Get and Set functions. Get takes two arguments that are interpreted as row and column indices, in that order. Set has three arguments, row and column indices followed by a new value. Coordinates are validated. If Get coordinates are bad, Get returns a zero. If Set coordinates are bad, no operation is performed. Otherwise, Get returns the indexed element and Set assigns a new value to the indexed element. The third operand of Get should be specified as 1 or 0. If any other number is specified, only the low-order bit will be used.

The two functions InsertRow and InsertCol can be used to overwrite an entire row or column with a vector of new values. The first operand of InsertRow and InsertCol is a vector of the proper dimension, and the second operand is an integer specifying the index of the row or column to be overwritten. If the index is invalid, no operation is performed. Otherwise the vector is copied into the specified row or column in order by index.

The functions GetRow and GetCol can be used to extract an entire row or column of values. GetRow and GetCol both have a single integer argument which is interpreted as the index of a
row or column. If the index is bad, a zero vector is returned, otherwise the elements of the indexed row are copied into the return value in order by index.

The classes Matrix3, Matrix4, and Matrix5 have demotion functions named Get2x2, Get3x3, and Get4x4. Get2x2 is implemented by all three classes, Get3x3 is implemented by Matrix4 and Matrix5, while Get4x4 is implemented only by Matrix5. Each of these functions has two arguments a row index followed by a column index. The functions return a matrix of the specified size whose upper left starts at the specified position. If an index is negative or too large to permit a matrix of the specified size to be returned, the identity matrix is returned instead.

(Note to self: Why no Insert2x2, Insert3x3, and Insert4x4?)

4.4 Utility Functions

The utility functions provide a means to perform most of the interesting functions that one would like to perform on a matrix. (But unfortunately, not all. There is no RANK function. There is no function that tells you the size of the smallest set of rows that add up to zero. There is no Row-Echelon function.)

4.4.1 Miscellaneous Functions

The following functions are, more or less, self explanatory. Test functions return an integer that is either one or zero

- Determinant(void) For GF(2) same as “NonSingular”
- Singular(void) True if matrix has no inverse.
- NonSingular(void) True if matrix has an inverse
- Transpose(void) Switch order of indices, return result
- isZero(void) True if every entry is zero
- isIdentity(void) True if this is the identity matrix.
- Invert(void) Returns inverse of matrix if any. Zero matrix otherwise.

The following functions are a bit more complex.

Order() returns zero if the matrix is singular, otherwise it returns the smallest integer \( n \) such that \( M^n = I \), where \( I \) is the identity matrix. For \( k \) dimensional matrices in GF(2) Order will never exceed \( 2^k - 1 \).

GetEigenPoly() This function returns an integer containing the eigenpolynomial or Characteristic polynomial of the matrix. Each bit of the integer represents one coefficient of the polynomial, with the low order bit representing the constant coefficient. Thus the value 19 represents the polynomial \( x^4 + x + 1 \). The degree of the polynomial will always match the dimension of the matrix. If the matrix is non-singular, the constant coefficient will be 1.

isPermutation() This is true if every row and every column of the matrix contains a single one and is false otherwise. True and false are indicated by an integer of 1 or 0 respectively.

GetPermutation() This function returns a string which must be deleted when no longer needed. If the matrix is singular, an empty string is returned. Otherwise, the nxn matrix is treated as a permutation on a set of \( 2^n - 1 \) elements (the non-zero input vectors) and is converted to cycle notation. If the matrix is the identity
the string “I” is returned. For example, the 3x3 matrix Matrix3(0,0,1, 0,1,0, 1,0,0) returns the string “(1,4) (3,6)”.  

Gaussian(Variables) This function performs Gaussian elimination on the matrix and a given vector of constants, and returns the solution to the linear system of equations, if any. The matrix is assumed to hold the variable coefficients, and the argument, Variables, must hold the constant values. Variables must be a vector of the same dimension as the matrix. The solution will overwrite the values in the Variables vector. The function returns 0 if the matrix is singular and 1 otherwise. If the matrix is singular, the Variables vector will contain garbage.

Minor(int x,int y) This function returns the determinant of the submatrix created by eliminating row x and column y from the current matrix. It is used to compute the determinants of the current matrix.

IntVal( ) This function converts an nxn matrix to an integer by concatenating the bits of each row into an n-bit number, and then by concatenating all n rows into a single integer. Concatenations are performed from left to right in ascending order by index.

4.4.2 Reducibilities.

The following functions are used to compute the reducibilities of a matrix. This actually only makes sense if we are talking about matrix groups, but these functions are necessary so they can be used by the Group classes. The idea of reducible and decomposable matrices is that a matrix can be decomposed into a direct sum of smaller matrices, which occur along the main diagonal, if certain matrix entries are zero.

Because conditions differ for different matrices, it is best to list the test functions class by class. In the following illustrations, the function will return true (1) if the indicated zero positions are zero in the matrix. Otherwise the function will return false (0). The positions marked by an x are not tested by the function.

4.4.3 Matrix2

Reducible1
\[
\begin{pmatrix}
  x & 0 \\
  x & x \\
\end{pmatrix}
\]

Reducible1
\[
\begin{pmatrix}
  x & x \\
  0 & x \\
\end{pmatrix}
\]

Decomposable
\[
\begin{pmatrix}
  x & 0 \\
  0 & x \\
\end{pmatrix}
\]

4.4.4 Matrix3

Reducible1
\[
\begin{pmatrix}
  x & 0 & 0 \\
  x & x & x \\
  x & x & x \\
\end{pmatrix}
\]
Reducible2
\[
\begin{pmatrix}
  x & x & 0 \\
  x & x & 0 \\
  x & x & x \\
\end{pmatrix}
\]

Reducible3
\[
\begin{pmatrix}
  x & x & x \\
  x & x & x \\
  0 & 0 & x \\
\end{pmatrix}
\]

Reducible4
\[
\begin{pmatrix}
  x & x & x \\
  0 & x & x \\
  0 & x & x \\
\end{pmatrix}
\]

Decomposable1
\[
\begin{pmatrix}
  x & 0 & 0 \\
  0 & x & x \\
  0 & x & x \\
\end{pmatrix}
\]

Decomposable1
\[
\begin{pmatrix}
  x & x & 0 \\
  x & x & 0 \\
  0 & 0 & x \\
\end{pmatrix}
\]

4.4.5 Matrix4

Reducible1
\[
\begin{pmatrix}
  x & 0 & 0 & 0 \\
  x & x & x & x \\
  x & x & x & x \\
  x & x & x & x \\
\end{pmatrix}
\]

Reducible2
\[
\begin{pmatrix}
  x & x & x & 0 \\
  x & x & x & 0 \\
  x & x & x & 0 \\
  x & x & x & x \\
\end{pmatrix}
\]

Reducible3
\[
\begin{pmatrix}
  x & x & x & x \\
  x & x & x & x \\
  x & x & x & x \\
  0 & 0 & 0 & x \\
\end{pmatrix}
\]

Reducible4
\[
\begin{pmatrix}
  x & x & x & x \\
  0 & x & x & x \\
  0 & x & x & x \\
  0 & x & x & x \\
\end{pmatrix}
\]
Reducible5
\[
\begin{pmatrix}
  x & x & 0 & 0 \\
  x & x & 0 & 0 \\
  x & x & x & x \\
  x & x & x & x \\
\end{pmatrix}
\]
Reducible6
\[
\begin{pmatrix}
  x & x & x & x \\
  x & x & x & x \\
  0 & 0 & x & x \\
  0 & 0 & x & x \\
\end{pmatrix}
\]
Decomposable1
\[
\begin{pmatrix}
  x & 0 & 0 & 0 \\
  0 & x & x & x \\
  0 & x & x & x \\
  0 & x & x & x \\
\end{pmatrix}
\]
Decomposable2
\[
\begin{pmatrix}
  x & x & x & 0 \\
  x & x & x & 0 \\
  x & x & x & 0 \\
  0 & 0 & 0 & x \\
\end{pmatrix}
\]
Decomposable3
\[
\begin{pmatrix}
  x & x & 0 & 0 \\
  x & x & 0 & 0 \\
  0 & 0 & x & x \\
  0 & 0 & x & x \\
\end{pmatrix}
\]

4.4.6 Matrix5
\[
\begin{pmatrix}
  x & 0 & 0 & 0 & 0 \\
  x & x & x & x & x \\
  x & x & x & x & x \\
  x & x & x & x & x \\
  x & x & x & x & x \\
\end{pmatrix}
\]
Reducible1
\[
\begin{pmatrix}
  x & x & x & x & 0 \\
  x & x & x & x & 0 \\
  x & x & x & x & 0 \\
  x & x & x & x & 0 \\
  x & x & x & x & x \\
\end{pmatrix}
\]
Reducible2
<table>
<thead>
<tr>
<th></th>
<th>Reduced3</th>
<th>Reduced4</th>
<th>Reduced5</th>
<th>Reduced6</th>
<th>Reduced7</th>
<th>Reduced8</th>
<th>Decomposable1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( \begin{pmatrix} x &amp; x &amp; x &amp; x &amp; x \ x &amp; x &amp; x &amp; x &amp; x \ x &amp; x &amp; x &amp; x &amp; x \ x &amp; x &amp; x &amp; x &amp; x \ 0 &amp; 0 &amp; 0 &amp; 0 &amp; x \end{pmatrix} )</td>
<td>( \begin{pmatrix} x &amp; x &amp; x &amp; x &amp; x \ 0 &amp; x &amp; x &amp; x &amp; x \ 0 &amp; x &amp; x &amp; x &amp; x \ 0 &amp; x &amp; x &amp; x &amp; x \end{pmatrix} )</td>
<td>( \begin{pmatrix} x &amp; x &amp; x &amp; x &amp; x \ x &amp; x &amp; x &amp; x &amp; x \ x &amp; x &amp; x &amp; x &amp; x \ x &amp; x &amp; x &amp; x &amp; x \end{pmatrix} )</td>
<td>( \begin{pmatrix} x &amp; x &amp; x &amp; 0 &amp; 0 \ x &amp; x &amp; x &amp; 0 &amp; 0 \ x &amp; x &amp; x &amp; 0 &amp; 0 \ x &amp; x &amp; x &amp; 0 &amp; 0 \ x &amp; x &amp; x &amp; x &amp; x \end{pmatrix} )</td>
<td>( \begin{pmatrix} x &amp; x &amp; x &amp; x &amp; x \ x &amp; x &amp; x &amp; x &amp; x \ x &amp; x &amp; x &amp; x &amp; x \ 0 &amp; 0 &amp; 0 &amp; x &amp; x \ 0 &amp; 0 &amp; 0 &amp; x &amp; x \end{pmatrix} )</td>
<td>( \begin{pmatrix} x &amp; x &amp; x &amp; x &amp; x \ x &amp; x &amp; x &amp; x &amp; x \ x &amp; x &amp; x &amp; x &amp; x \ 0 &amp; 0 &amp; x &amp; x &amp; x \ 0 &amp; 0 &amp; x &amp; x &amp; x \end{pmatrix} )</td>
<td>( \begin{pmatrix} x &amp; 0 &amp; 0 &amp; 0 &amp; 0 \ 0 &amp; x &amp; x &amp; x &amp; x \ 0 &amp; x &amp; x &amp; x &amp; x \ 0 &amp; x &amp; x &amp; x &amp; x \ 0 &amp; x &amp; x &amp; x &amp; x \end{pmatrix} )</td>
</tr>
</tbody>
</table>
Decomposable 2
\[
\begin{pmatrix}
  x & x & x & x & 0 \\
  x & x & x & x & 0 \\
  x & x & x & x & 0 \\
  x & x & x & x & 0 \\
  0 & 0 & 0 & 0 & x
\end{pmatrix}
\]

Decomposable 3
\[
\begin{pmatrix}
  x & x & 0 & 0 & 0 \\
  x & x & 0 & 0 & 0 \\
  0 & 0 & x & x & x \\
  0 & 0 & x & x & x \\
  0 & 0 & x & x & x
\end{pmatrix}
\]

Decomposable 4
\[
\begin{pmatrix}
  x & x & x & 0 & 0 \\
  x & x & x & 0 & 0 \\
  x & x & x & 0 & 0 \\
  x & x & x & 0 & 0 \\
  0 & 0 & 0 & x & x \\
  0 & 0 & 0 & x & x
\end{pmatrix}
\]

4.4.7 Extracting Subcomponents
Once a matrix is determined to be either reducible or decomposable, it is usually desirable to extract the component matrices. Again, this is different depending on matrix size, so different functions are implemented for each dimension. The following describes the functions implemented for each class.

4.4.8 Matrix 2
GetLeftComponent( ) Returns element 0,0.
GetRightComponent( ) Returns element 1,1

4.4.9 Matrix 3
GetLeftComponent( ) Returns the 2x2 matrix with upper left corner at (0,0).
GetRightComponent( ) Returns the 2x2 matrix with upper left corner at (1,1).

4.4.10 Matrix 4
GetLeftComponent( ) Returns the 3x3 matrix with upper left corner at (0,0).
GetRightComponent( ) Returns the 3x3 matrix with upper left corner at (1,1).
GetLeft2x2( ) Returns the 2x2 matrix with upper left corner at (0,0).
GetRight2x2( ) Returns the 2x2 matrix with upper left corner at (2,2).

4.4.11 Matrix 5
GetLeftComponent( ) Returns the 4x4 matrix with upper left corner at (0,0).
GetRightComponent( ) Returns the 4x4 matrix with upper left corner at (1,1).
GetLeft3x3( ) Returns the 3x3 matrix with upper left corner at (0,0).
GetRight3x3( ) Returns the 3x3 matrix with upper left corner at (2,2).
GetLeft2x2( ) Returns the 2x2 matrix with upper left corner at (0,0).
GetRight2x2( ) Returns the 2x2 matrix with upper left corner at (3,3).
4.4.12 Serialization Functions

Serialization functions are used primarily for printing. Each of these functions returns a newly allocated string that must be deleted once it is no longer needed. The following functions are available.

- **Serialize( )** convert the matrix to a string of ones and zeros with newlines between the rows. Row 0 will appear on top and column 0 will appear at the left.
- **SerializeNL( )** Same as Serialize, but with an extra newline at the end of the string.
- **SerializeRow(Row)** Convert the specified row of the matrix into a string with no newline at the end. (or anywhere else in the string). The entry for column zero appears first. If the index is invalid an empty string (not NULL) is returned.
- **SerializeRowNL(Row)** Same as SerializeRow, but with a newline at the end of the string. The newline will always be present even if the string is empty.

5 Collection Types

The collection types specify a standard set of functions which will be discussed in this section. Those functions that are unique to a particular class will be discussed in succeeding sections. The collection types are all linked lists of other types. A VectorList is a list of Vectors, an Orbit is a list of VectorLists, a Group is a list of matrices, and a GroupList is a list of groups. The Orbit and GroupList objects have NEXT pointers and can be formed into custom lists if needed.

5.1 Constructors

Each collection class has a default constructor, copy constructor, destructor and assignment overload. These work as expected, except the NEXT pointer is never overwritten. (This is also true for Vector and Matrix assignments.)

Each collection class has constructors that permit singleton collections to be created. The operand may be either an object of the contained type or a pointer to an object of the contained type. These constructors cannot generally be used as type converters because of the double-conversion rule. They can, however, be explicitly specified to force a type conversion. The Orbit and GroupList classes have constructors that permit them to be constructed from Vectors and Matrices respectively. The argument may be either an object or a pointer to an object. A singleton VectorList may also be created using a list of values for the single vector element as in the following examples.

```c
VectorList2 VL2(1,1);
VectorList3 VL3(1,1,1);
VectorList4 VL4(1,1,1,1);
VectorList5 VL5(1,1,1,1,1);
```

The classes of dimensions 3, 4, and 5 have promotion constructors that permit a collection to be created from any lower-dimension object of the same kind.
5.2 Operator Overloads

Each collection class overloads all six comparison operators. For two lists to be equal, they must be the same length, and the objects in the corresponding positions of the list must compare equal. If two collections are identical, but sorted into different orders, they will not compare equal. If two collections are the same, but one has duplicated elements and the other does not, they will not compare the same. The RmDups function must be used on collections that are to be compared as sets. This function will sort the lists into ascending order and remove all duplicate elements.

When two lists are not equal, the first differing position determines the order of the lists. The comparison operator of the contained type is used to determine the order. If there is no differing position, then the shorter list is the smaller.

The + and += operators are used to form the union of two collections. The result is not a true union, but the concatenation of two lists. If any element occurs in both lists, it will be duplicated in the result of + or +=. To form the true union, + or += must be followed by a call to the RmDups function.

The * and *= operators are used to form the true intersection. The result will be a sorted unduplicated list containing all items that occur in both collections.

The binary operators – and -= are used to form the set difference of two collections. The result will be a sorted unduplicated collection of items from the first operand, with all the items that occur in the second operand removed.

The unary operator – is used to obtain the true set complement. It is not implemented for some classes, because the universal set is too large.

Groups implement the additional operators ^ and ^=. These are used to obtain the group-theoretic union of two matrix groups (with respect to matrix multiplication). The result of the operation A^B will be the smallest matrix group containing both A and B. (Strictly speaking, ^ returns a multiplicative set, because Group objects can contain singular matrices.)

5.3 Accessors

Accessors make the various parts of a collection available. The most important function is Add, which takes a single argument which must be an object of the contained type. This object is copied and the copy is added to the collection. This function is used as a utility function by other member functions, so all additions to the collection must pass through this function.

VectorLists provide an additional Add function which has a list of integer arguments as in the following examples.

VectorList2( ).Add(1,1);
VectorList3( ).Add(1,1,1);
VectorList4( ).Add(1,1,1,1);
VectorList5( ).Add(1,1,1,1,1);

Each collection class has a pair of enumerators that can be used to go through each item in the collection. The names of these functions are different depending on the class. The following lists the enumerators for the four kinds of collections.

VectorList
   GetFirstVector( )
GetNextVector( )

Orbit
  GetFirstList( )
  GetNextList( )

Group
  GetFirstMatrix( )
  GetNextMatrix( )

GroupList
  GetFirstGroup( )
  GetNextGroup( )

These functions return pointers to objects, and will return NULL when the list is exhausted. The following is an example of how to use these enumerators.

Group4 G;
...
for (Matrix4 * Temp = G.GetFirstMatrix( ) ; Temp ; Temp = G.GetNextMatrix( ))
{
  cout<<Temp->Order( )<<endl;
}

The GetMatrix(int) function can be used to start the enumeration at some place other than the beginning. Thus the following example starts the enumeration with the third element of the list. The first element of the list is element zero.

Group4 G;
...
for (Matrix4 * Temp = G.GetMatrix(2) ; Temp ; Temp = G.GetNextMatrix( ))
{
  cout<<Temp->Order( )<<endl;
}

For VectorLists, Orbits, and GroupLists, the functions are GetVector(int), GetList(int), and GetGroup(int). These functions should not be used exclusively to enumerate lists, because each call searches the list from the beginning.

Items can be deleted from the list using their pointers. The most common way to do this is to first use the GetVector(int), GetList(int), GetMatrix(int), or GetGroup(int) function to get a pointer to the proper element, and then use the return value of this function in the Delete function, as in the following example.

// delete the fifth vector
Int Successful = Delete(GetVector(4));
The Delete function returns 1 or 0 to indicate whether the deletion was successful. 1 indicates success. The object is removed from the collection and destroyed.

The function Order() returns the number of elements in the collection.

Each collection object has an internal NEXT pointer that permits it to be added to a higher-level collection. The functions GetNext and SetNext provide access to this pointer. GetNext returns a pointer which is the value of the internal NEXT pointer. SetNext has a single argument which is assigned to the NEXT pointer. The argument of SetNext is not audited in any way. (The NEXT pointer should probably be public, but I thought I’d try it this way for a change.)

The 3, 4, and 5 dimensional collections have downgrade functions that permit the objects to be downgraded to a lower dimension. These functions are based on the downgrade functions provided by the Matrix and Vector classes, and result in the Matrix or Vector function being applied to each member of the class, perhaps recursively.

The downgrade functions implemented by each class are as follows.

**VectorList3**
- GetVL2(int)

**VectorList4**
- GetVL2(int)
- GetVL3(int)

**VectorList5**
- GetVL2(int)
- GetVL3(int)
- GetVL4(int)

**Orbit3**
- GetO2(int)

**Orbit4**
- GetO2(int)
- GetO3(int)

**Orbit5**
- GetO2(int)
- GetO3(int)
- GetO4(int)

**Group3**
- Get2x2(int,int)

**Group4**
- Get2x2(int,int)
- Get3x3(int,int)
Group5
Get2x2(int,int)
Get3x3(int,int)
Get4x4(int,int)

GroupList3
Get2x2(int,int)

GroupList4
Get2x2(int,int)
Get3x3(int,int)

GroupList5
Get2x2(int,int)
Get3x3(int,int)
Get4x4(int,int)

5.4 Utility Functions
Each collection object provides a number of utility functions, most of which are more or less self-explanatory.

Sort    Sorts the objects into ascending order using the comparison operators of the objects.
Clear   Deletes all objects in the collection.
HasDups Returns 1(true) or 0(false) indicating whether or not the collection contains duplicate elements. This is an n-squared search so use it with caution.
RmDups  Sorts the collection into ascending order and removes all duplicate items.
GenerateAll Generates the universal set for the collection, where possible. This function is not implemented if the universal set is overly large. For example GroupList5::GenerateAll( ) is not implemented because if we were to write down the number of elements in this set, the number would have over a million digits.
Serialize This function returns a string containing a printable version of the object. This string must be deleted when it is no longer needed.
SerializeNL Same as Serialize, but with an additional end of line character at the end of the string.
Contains(Object) Returns true (1) or false (0) depending on whether the collection contains an object that is equal to the argument.

Vectors contain two additional functions, a Contains function that allows one to specify the vector as a list of integers, and a GenerateAll(int) function that permits the generation of all vectors of a particular weight.

6 Class-Specific Functions
In addition to the standard functions implemented by every collection class, there are certain functions that are implemented only by a specific class. These functions are found exclusively in the Group and GroupList classes.
All Group classes implement the following functions.

void Close( ) Computes the algebraic closure with respect to matrix multiplication. It is assumed that the initial collection contains no duplicates. If this isn’t true, the results could be a little weird. In most cases, this function is used to build a group from a small set of generators.

int Close(int) Computes the algebraic closure of a group. If the limit is exceeded, the closure computation is aborted, and the resultant collection should be discarded. If this happens, the function returns a zero. The function also returns a zero if the closure process stops on its own, but fails to construct a group of the specified size. The primary use of this function is to go through a large set of potential generators looking for groups isomorphic to a specific group. The first step in this process is to find a group of the right size, and then verify the isomorphism (usually manually).

Groupn Conjugate(Matrixn) This function computes the conjugate of a group and returns the result as a new group. The original group is not modified. Given a matrix $M$ and a non-singular matrix $T$, the conjugate of $M$ with respect to $T$ is the matrix $TMT^{-1}$. The conjugate of a matrix group $G$ with respect to a non-singular matrix $T$ is found by computing the conjugate with respect to $T$ of every element of $G$. The result is a new group which may be identical to the old group, but usually isn’t. Mathematically, conjugate groups are considered to be “different versions” of the same group.

void GenerateGLG(void) Generate the General Linear Group for a particular dimension. The new group replaces the existing one, if any. There are two important matrix groups of nxn matrices, the General Linear Group which contains all non-singular nxn matrices, and the Special Linear Group, which contains all matrices of determinant 1. In GF(2) these groups are the same. The algorithm is rather interesting. For nxn matrices, n nested loops are required. The algorithm maintains a list of all n-element non-zero vectors. The outer loop chooses vectors one at a time and marks the current vector as “in use.” The next loop chooses an unmarked vector and marks it as “in use”. It also marks the sum of the two “in use” vectors as “in use”. Inner loops become increasingly more complex as more complex linear combinations need to be marked as “in use.” Once a loop terminates, it unmarks all vectors that it marked when it began. Each chosen vector becomes a row of the generated matrix. The number of nxn non-singular matrices is given by the formula $\prod_{i=0}^{n/2} (2^n - 2^i)$.

Void GenerateOrder(int) The argument is interpreted as the order of a matrix. This function is the same as GenerateGLG, except only those matrices of the specified order are generated. If there are no matrices of that order, the resultant group is empty. Note that the order of a matrix $M$ is the smallest integer $n$ such that $M^n = I$ where $I$ is the identity matrix.

Orbitn GetOrbits(void) This function returns the orbits of a particular group. Two vectors, $V$ and $W$ are in the same orbit of a group $G$, if there is a matrix $M \in G$ such that $V \times M = W$. The Orbit returned is a list of VectorLists with each orbit represented as a separate VectorList.
Orbit2 GetVOrbits(void) The same as GetOrbits except the vectors are treated as column vectors and the multiplication is done in the order $M \times V = W$. This operation is the same as transposing every member of a group and computing the orbits.

All GroupList objects implement the CloseAll(void) function, which performs the Close operation on each member group.

6.1 Group2 Functions
The specialized Group2 functions generate certain special groups and permit the General Linear Group to be split into subsets based on order.

GenerateS2 This function generates the following set of matrices, which is the standard representation of $S_2$, the symmetric group of order 2.

\[
\begin{pmatrix}
1 & 0 \\
0 & 1 \\
1 & 0
\end{pmatrix}
\begin{pmatrix}
0 & 1 \\
1 & 0
\end{pmatrix}
\]

GenerateS3 This function generates the following set of matrices, which is a 2x2 representation of $S_3$, the symmetric group of order 3. (This is also the General Linear Group of 2x2 matrices.)

\[
\begin{pmatrix}
1 & 0 \\
0 & 1 \\
1 & 0
\end{pmatrix}
\begin{pmatrix}
0 & 1 \\
1 & 0 \\
0 & 1
\end{pmatrix}
\begin{pmatrix}
1 & 0 \\
0 & 1 \\
1 & 0
\end{pmatrix}
\begin{pmatrix}
1 & 0 \\
1 & 1 \\
1 & 1
\end{pmatrix}
\begin{pmatrix}
0 & 1 \\
1 & 1 \\
1 & 1
\end{pmatrix}
\begin{pmatrix}
1 & 1 \\
1 & 0 \\
0 & 1
\end{pmatrix}
\]

void GLGSplit(Group2 &O1,Group2 &O2,Group2 &O3) This function splits the general linear group into order1, order2 and order3 matrices. The three argument groups are cleared and the appropriate matrices are then added to them. Note the call by reference. The group itself is cleared and replaced with the entire General Linear Group.

6.2 GroupList2 Functions
GenerateS2Conj This function generates all conjugates of the group generated by GenerateS2. These groups are as follows.

\[
\begin{pmatrix}
1 & 0 \\
0 & 1 \\
1 & 0
\end{pmatrix}
\begin{pmatrix}
0 & 1 \\
1 & 0
\end{pmatrix}
\begin{pmatrix}
1 & 0 \\
0 & 1 \\
1 & 0
\end{pmatrix}
\begin{pmatrix}
0 & 1 \\
1 & 0 \\
1 & 0
\end{pmatrix}
\begin{pmatrix}
1 & 0 \\
0 & 1 \\
1 & 0
\end{pmatrix}
\begin{pmatrix}
1 & 0 \\
0 & 1 \\
1 & 0
\end{pmatrix}
\]

6.3 Group3 Functions
6.3.1 GenerateS2
GenerateS2 embeds the standard representation of $S_2$ into a set of 3x3 matrices as follows.
6.3.2 GenerateS3

GenerateS3 generates the standard representation of $S_3$, which is given below.

\[
\begin{pmatrix}
  1 & 0 & 0 \\
  0 & 1 & 0 \\
  0 & 0 & 1 \\
\end{pmatrix}
\begin{pmatrix}
  0 & 1 & 0 \\
  1 & 0 & 0 \\
  0 & 1 & 0 \\
\end{pmatrix}
\begin{pmatrix}
  0 & 0 & 1 \\
  1 & 0 & 0 \\
  0 & 0 & 1 \\
\end{pmatrix}
\begin{pmatrix}
  0 & 1 & 0 \\
  1 & 0 & 0 \\
  0 & 1 & 0 \\
\end{pmatrix}
\begin{pmatrix}
  0 & 0 & 1 \\
  1 & 0 & 0 \\
  0 & 0 & 1 \\
\end{pmatrix}
\begin{pmatrix}
  0 & 1 & 0 \\
  0 & 0 & 1 \\
  0 & 1 & 0 \\
\end{pmatrix}
\]

6.3.3 GenerateS4H and GenerateS4V

GenerateS4H and GenerateS4V both generate groups that are isomorphic to $S_4$, the symmetric group of order 4. The technique is to add either a row or column of all ones to the standard representation.

To be more specific, GenerateS4H generates the set of all matrices that can be created out of the following four rows.

100
010
001
111

By the same token Generate S4V generates the set of all matrices that can be created out of the following four columns.

1 0 0 1
0 1 0 1
0 0 1 1
0 1 1 1

6.3.4 GLGSplit

The following function splits the 3x3 general linear group into disjoint sets based on matrix order. There are no 3x3 matrices of orders 5 or 6, but these arguments are supplied anyway to relieve the user of remembering this fact. Each of the arguments is cleared. Then each argument is filled with all 3x3 matrices of the appropriate order. The group itself is cleared and replaced with the general linear group.

```c
GLGSplit( Group3 &O1,
        Group3 &O2,
        Group3 &O3,
        Group3 &O4,
        Group3 &O5,
        Group3 &O6,
        Group3 &O7)
```

6.4 GroupList3 Functions

GroupList3 provides the following four functions which are used to generate all conjugates of the groups generated by Group3 functions.

- GenerateS2Conj
- GenerateS3Conj
- GenerateS4HConj
- GenerateS4VConj

6.5 Group4 functions

6.5.1 GenerateS2

GenerateS2 erases the existing group, if any, and replaces it with a group that is isomorphic to \( S_2 \). Specifically, the following group.

\[
\begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\]

6.5.2 GenerateS2A

There are two conjugacy classes of \( S_2 \) isomorphs in the 4x4 general linear group. GenerateS2 generates a member of one of these classes, GenerateS2A generates a member of the other class. Specifically, the following group.

\[
\begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0
\end{pmatrix}
\]

\[
\text{Matrix4}(1,0,0,0, 0,1,0,0, 0,0,1,0, 0,0,0,1),
\text{Matrix4}(0,0,0,1, 0,0,1,0, 0,1,0,0, 1,0,0,0)
\]

6.5.3 GenerateS3

GenerateS3 erases the existing group, if any, and replaces it with a group that is isomorphic to \( S_3 \). Specifically, the following group.

\[
\begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1
\end{pmatrix}
\]
6.5.4 GenerateS3A and GenerateS3B

There are two conjugacy classes of $S_3$ isomorphs in the 4x4 general linear group. GenerateS3 generates a member of one of these classes, GenerateS2A and GenerateS2B generate a members of the other classes. Specifically, they generate the following groups.

**GenerateS3A**

\[
\begin{pmatrix}
0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 \\
1 & 0 & 0 & 1 \\
\end{pmatrix}
\]

**GenerateS3B**

\[
\begin{pmatrix}
0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 \\
1 & 0 & 0 & 1 \\
\end{pmatrix}
\]

6.5.5 GenerateS4

GenerateS4(void) erases the existing group, if any, and replaces it with the standard representation of $S_4$. Specifically, this is the set of all 4x4 permutation matrices.

6.5.6 GenerateS4A-GenerateS4H

In addition to the standard representation of $S_4$, there are many other 4x4 matrix groups that are isomorphic to $S_4$. There are 420 conjugates of the standard representation (including the standard representation itself) and eight other conjugacy classes of matrices of various sizes. The functions GenerateS4A, GenerateS4B, GenerateS4C, GenerateS4D, GenerateS4E, GenerateS4F, GenerateS4G, and GenerateS4H can be used to generate one group from each of these classes. The generators for each of these groups are given below.

**Generate4A**

\[
\begin{pmatrix}
0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 \\
1 & 0 & 1 & 0 \\
\end{pmatrix}
\]

**Generate4B**

\[
\begin{pmatrix}
0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 \\
1 & 0 & 1 & 0 \\
\end{pmatrix}
\]

\[
\begin{pmatrix}
0 & 0 & 1 & 1 \\
0 & 1 & 1 & 0 \\
0 & 1 & 0 & 1 \\
0 & 0 & 1 & 0 \\
\end{pmatrix}
\]
<table>
<thead>
<tr>
<th>Generates4C</th>
<th>0 0 0 1</th>
<th>1 0 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 0 1 0</td>
<td>0 0 1 0</td>
</tr>
<tr>
<td></td>
<td>0 1 0 0</td>
<td>0 1 1 0</td>
</tr>
<tr>
<td></td>
<td>1 0 1 0</td>
<td>0 0 1 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Generates4D</th>
<th>0 0 0 1</th>
<th>0 0 0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 0 1 0</td>
<td>0 0 1 1</td>
</tr>
<tr>
<td></td>
<td>0 1 0 0</td>
<td>0 1 0 1</td>
</tr>
<tr>
<td></td>
<td>1 1 1 0</td>
<td>1 1 1 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Generates4E</th>
<th>0 0 0 1</th>
<th>0 0 0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 0 1 0</td>
<td>0 1 0 0</td>
</tr>
<tr>
<td></td>
<td>0 1 0 0</td>
<td>0 0 1 0</td>
</tr>
<tr>
<td></td>
<td>1 1 1 0</td>
<td>1 0 0 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Generates4F</th>
<th>0 0 0 1</th>
<th>1 0 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 0 1 0</td>
<td>0 0 1 0</td>
</tr>
<tr>
<td></td>
<td>0 1 0 0</td>
<td>1 0 1 0</td>
</tr>
<tr>
<td></td>
<td>1 1 1 0</td>
<td>1 1 1 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Generates4G</th>
<th>0 0 0 1</th>
<th>0 0 0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 0 1 0</td>
<td>0 0 1 0</td>
</tr>
<tr>
<td></td>
<td>0 1 0 0</td>
<td>1 0 1 1</td>
</tr>
<tr>
<td></td>
<td>1 1 1 0</td>
<td>1 1 1 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Generates4H</th>
<th>0 0 0 1</th>
<th>0 0 0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 0 1 0</td>
<td>0 0 1 0</td>
</tr>
<tr>
<td></td>
<td>0 1 0 0</td>
<td>0 1 1 0</td>
</tr>
<tr>
<td></td>
<td>1 0 1 0</td>
<td>1 0 0 1</td>
</tr>
</tbody>
</table>
6.5.7 GenerateS5V and GenerateS5H

GenerateS5V and GenerateS5H erase any existing group and replace it with groups isomorphic to \( S_5 \). The two groups generated by these functions are conjugate to one another. GenerateS5H adds a row of all 1’s to the standard representation, while GenerateS5V adds a column of all ones. That is to say, GenerateS5H generates all matrices that can be created from the five following rows.

\[
\begin{align*}
0001, & \quad 0010, \quad 0100, \quad 1000, \quad 1111.
\end{align*}
\]

By the same token, GenerateS5V generates all the matrices that can be constructed from the following 5 columns.

\[
\begin{align*}
0 & \quad 0 & \quad 0 & \quad 1 & \quad 1 \\
0 & \quad 0 & \quad 1 & \quad 0 & \quad 1 \\
0 & \quad 1 & \quad 0 & \quad 0 & \quad 1 \\
1 & \quad 0 & \quad 0 & \quad 0 & \quad 1
\end{align*}
\]

6.5.8 GenerateS5A

There are two conjugacy classes of \( S_5 \) isomorphs. The functions GenerateS5V and GenerateS5H generate members of one class, GenerateS5A generates a member of the other class. The generators of this group are as follows.

\[
\begin{align*}
\begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}, & \quad \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}
\end{align*}
\]

6.5.9 GLGSplit

The following function splits the 4x4 general linear group into disjoint sets based on matrix order. A number of the argument groups will be empty when the function returns, but these arguments are supplied anyway to relieve the user of remembering which are empty and which are not. (There are no 4x4 matrices of orders 8-14.) Each of the arguments is cleared. Then each argument is filled with all 4x4 matrices of the appropriate order. The group itself is cleared and replaced with the general linear group.

\[
\text{GLGSplit( Group4 &O1, Group4 &O2, Group4 &O3, Group4 &O4, Group4 &O5, Group4 &O6, Group4 &O7, Group4 &O8, Group4 &O9, Group4 &O10,}
\]
6.6 GroupList4 Functions
The specialized GroupList4 functions are used to generate the conjugates of the groups generated by the special functions of the Group4 class. The following functions are available.

- GenerateS2Conj
- GenerateS2AConj
- GenerateS3Conj
- GenerateS3AConj
- GenerateS3BConj
- GenerateS4Conj
- GenerateS4AConj
- GenerateS4BConj
- GenerateS4CConj
- GenerateS4DConj
- GenerateS4EConj
- GenerateS4FConj
- GenerateS4GConj
- GenerateS4HConj
- GenerateS5HConj
- GenerateS5VConj
- GenerateS5AConj

6.7 Group5 Functions

6.7.1 GenerateS2 – GenerateS4
The functions GenerateS2 through GenerateS4 erase the existing group, if any, and replace it with an isomorphic copy of $S_2$, $S_3$, or $S_4$, respectively. The way these groups are constructed is first the standard 2x2, 3x3, or 4x4 representation is constructed, and then this is embedded in a set of 5x5 matrices. The smaller matrices will be placed in the upper left corner of the 5x5 identity matrix.

6.7.2 GenerateS5
This function erases the current group, if any, and replaces it with the standard representation of $S_5$. In other words, this is the set of all 5x5 permutation matrices.
6.7.3 GenerateS5A-GenerateS5D

There are four conjugacy classes of $S_5$ isomorphs in the 5x5 general linear group. The functions GenerateS5A, GenerateS5B, GenerateS5C, and GenerateS5D generate a member of each of these four classes. The group generated by GenerateS5A is conjugate to the standard representation. The generators of these groups are given below.

**GenerateS5A**

\[
\begin{pmatrix}
0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 0
\end{pmatrix}
\begin{pmatrix}
0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 \\
1 & 0 & 0 & 1 & 1
\end{pmatrix}
\]

**GenerateS5B**

\[
\begin{pmatrix}
0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 0
\end{pmatrix}
\begin{pmatrix}
0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 1
\end{pmatrix}
\]

**GenerateS5C**

\[
\begin{pmatrix}
0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 0
\end{pmatrix}
\begin{pmatrix}
0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 1
\end{pmatrix}
\]

**GenerateS5D**

\[
\begin{pmatrix}
0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 0
\end{pmatrix}
\begin{pmatrix}
0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 1
\end{pmatrix}
\]

6.7.4 Generate6H and Generate6V

Generate6H and Generate6V generate groups of 5x5 matrices that are conjugate to $S_6$. Generate6H adds a row of all ones to the standard representation, while Generate6V adds a column of all ones to the standard representation. In other words, Generate6H generates all matrices that can be constructed from the following rows.
By the same token, Generate6V generates all matrices that can be generated from the following six columns. The groups generated by Generate6H and Generate6V are not conjugate to one another.

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

### 6.7.5 GLGSplit

The following function splits the 5x5 general linear group into disjoint sets based on matrix order. This function requires a 64-bit operating system, and has not been tested. A number of the argument groups will be empty when the function returns, but these arguments are supplied anyway to relieve the user of remembering which are empty and which are not. (There are no 5x5 matrices of orders 9, 10, 11, 13, 16-20, and 22-30.) Each of the arguments is cleared. Then each argument is filled with all 5x5 matrices of the appropriate order. The group itself is cleared and replaced with the general linear group.

```c
GLGSplit( Group5 &O1, Group5 &O2,
         Group5 &O3, Group5 &O4,
         Group5 &O5, Group5 &O6,
         Group5 &O7, Group5 &O6,
         Group5 &O8, Group5 &O9,
         Group5 &O10, Group5 &O11,
         Group5 &O12, Group5 &O13,
         Group5 &O14, Group5 &O15,
         Group5 &O16, Group5 &O17,
         Group5 &O18, Group5 &O19,
         Group5 &O20, Group5 &O21,
         Group5 &O22, Group5 &O23,
         Group5 &O24, Group5 &O25,
         Group5 &O26, Group5 &O27,
         Group5 &O28, Group5 &O29,
         Group5 &O30, Group5 &O31)
```

### 6.8 GroupList5 Functions

The GroupList5 functions are used to generate all conjugates of the functions generated by the specialized Group5 functions. The following functions are available.

- GenerateS2Conj
- GenerateS3Conj
- GenerateS4Conj
The Generator Classes

The generator classes are provided to generate all members of the General Linear Group without storing them. The GenerateGLG Group functions store all members of the group. This can cause problems, especially for 5x5 matrices. The generators are especially intended for these large matrices, but are provided for all matrices as a convenience.

Additional classes are provided to generate all matrices of a certain size, but these classes are not necessary since a loop like the following will do the same thing.

```cpp
for (int i=0 ; i<(1<<25) ; i++)
{
    Matrix5 M(i);
    ...
}
```

The generator classes are derived publicly from the respective matrix classes, as follows.

```cpp
class GenerateGLG2: public Matrix2
class GenerateGLG3: public Matrix3
class GenerateGLG4: public Matrix4
class GenerateGLG5: public Matrix5

class GenerateAll2: public Matrix2
class GenerateAll3: public Matrix3
class GenerateAll4: public Matrix4
class GenerateAll5: public Matrix5
```

Where necessary, type conversions are provided between the generator classes and the matrix classes so that a generator object can be used, pretty much, anywhere a matrix object can be used.

Each generator class implements the default constructor, copy constructor, destructor, and assignment overload. There is also a type-conversion constructor that converts a matrix to a generator of the same dimension.

In addition, the following utility functions are implemented.

```cpp
int First( ) // This function resets the base matrix to the first matrix in the set and returns a 1.
```
int Next() This function converts the base Matrix into the next element in the set and returns a 1. If the set is exhausted, the base matrix is not changed and a zero is returned.

int First(int) The operand is considered to represent a matrix order. This function resets the base matrix of the first order and returns a 1. If there is no matrix of the specified order, the base matrix is garbage and a zero is returned.

int Next(int) This function converts the base Matrix into the next matrix of the specified order and returns a 1. If there are no more matrices of the specified order, the base matrix is garbage and the function returns a 0. Although it is the usual practice for First(int) and Next(int) to have the same value for their arguments, this is not required. It is also possible to mix First(int), and Next(int) with First() and Next().

8 Conclusion

This package is provided with no warranty of any kind. Report any bugs, problems, or suggestions for improvement to Peter M. Maurer at Peter_Maurer@Baylor.edu.

9 Appendix: The Class Definitions

The following section contains the class definitions of each class. If arguments and return values are not clear from the discussion above, this is the authoritative source.

9.1 Vector2
class Vector2
{
public:
    // constructors
    Vector2();
    Vector2(const Vector2 &x);
    Vector2(int x0, int x1);
    // type converters
    Vector2(int xAll);
    operator int() { return IntVal(); }
    Vector2(char *x);
    operator char*() { char * rv = new char[3]; rv[0] = Data[0] + '0'; rv[1] = Data[1] + '0'; rv[2] = '\0'; return rv; }
    // destructor
    ~Vector2();
    // operator overloads
    Vector2 &operator=(const Vector2 &x);
    Vector2 &operator+=(Vector2 &x);
    Vector2 &operator*=(Matrix2 &x);
    Vector2 &operator*=(int x);
    Vector2 operator+(Vector2 &x);
    int operator*(Vector2 &x);
    Vector2 operator*(int x);
    Vector2 operator*(Matrix2 &x);
    int operator-=(Vector2 &x);
    int operator!=(Vector2 &x);
    int operator<(Vector2 &x);
    int operator<=(Vector2 &x);
    int operator>(Vector2 &x);
    int operator>=(Vector2 &x);
int operator>=(Vector2 &x);
// accessors
Vector2 * GetNext();
void SetNext(Vector2 * NewNext);
int Get(int x);
void Set(int x, int NewVal);
int Order(void);
// utilities
int isZero(void);
int isOne(void);
char * Serialize(void);
char * SerializeNL(void);
int IntVal(void);
protected:
    Vector2 * Next;
    int Data[2];
};

9.2 VectorList2

class VectorList2
{
    friend VectorList3;
    friend VectorList4;
    friend VectorList5;
public:
    // constructors
    VectorList2();
    VectorList2(const VectorList2 &x);
    VectorList2(Vector2 &x);
    VectorList2(Vector2 *x);
    VectorList2(int x, int y);
    // destructor
    ~VectorList2();
    // operator overloads
    VectorList2 &operator=(const VectorList2 &x);
    VectorList2 &operator+=(const VectorList2 &x);
    VectorList2 &operator*=(VectorList2 &x);
    VectorList2 &operator-=(VectorList2 &x);
    VectorList2 operator+(const VectorList2 &x); // union
    VectorList2 operator*(VectorList2 &x); // intersection
    VectorList2 operator-(VectorList2 &x); // set difference
    VectorList2 operator-(); // complement
    int operator==(VectorList2 &x);
    int operator!=(VectorList2 &x);
    int operator<(VectorList2 &x);
    int operator<=(VectorList2 &x);
    int operator>(VectorList2 &x);
    int operator>=(VectorList2 &x);
    // accessors
    Vector2 * GetFirstVector(void);
    Vector2 * GetNextVector(void);
    Vector2 * GetVector(int x);
    VectorList2 * GetNext(void);
    void SetNext(VectorList2 * NewNext);
    void Add(Vector2 &x);
    void Add(int x, int y);
int Delete(Vector2 *x);
int Order(void);

// utilities
void Sort(void);
void Clear(void);
char *Serialize(void);
char *SerializeNL(void);
int Contains(Vector2 &x);
int Contains(int x, int y);
int HasDups(void);
void RmDups(void);
void GenerateAll(void);
void GenerateAll(int x);

protected:
VectorList2 *Next;
Vector2 *Head;
Vector2 *Tail;
int Count;
Vector2 *Current;

};

9.3 Orbit2

class Orbit2
{
 friend Orbit3;
 friend Orbit4;
 friend Orbit5;
 public:

 // constructors
 Orbit2();
 Orbit2(const Orbit2 &x);
 Orbit2(VectorList2 &x);
 Orbit2(VectorList2 *x);
 Orbit2(Vector2 &x);
 Orbit2(Vector2 *x);

 // destructor
 ~Orbit2();

 // operator overloads
 Orbit2 &operator=(const Orbit2 &x);
 Orbit2 &operator+=(const Orbit2 &x);
 Orbit2 &operator*=(Orbit2 &x);
 Orbit2 &operator-=(Orbit2 &x);
 Orbit2 operator+(const Orbit2 &x); // union
 Orbit2 operator*(Orbit2 &x); // intersection
 Orbit2 operator-(Orbit2 &x); // set difference
 Orbit2 operator-(void); // complement

 int operator==(Orbit2 &x);
 int operator!=(Orbit2 &x);
 int operator<(Orbit2 &x);
 int operator<=(Orbit2 &x);
 int operator>(Orbit2 &x);
 int operator>=(Orbit2 &x);

 // accessors
 VectorList2 * GetFirstList(void);
 VectorList2 * GetNextList(void);
 VectorList2 * GetList(int x);
Orbit2 * GetNext(void);
void SetNext(Orbit2 * NewNext);
void Add(VectorList2 &x);
int Delete(VectorList2 *x);
int Order(void);
// utilities
void Sort(void);
void Clear(void);
char *Serialize(void);
char *SerializeNL(void);
int Contains(VectorList2 &x);
int HasDups(void);
void RmDups(void);
void GenerateAll(void);
protected:
Orbit2 *Next;
VectorList2 *Head;
VectorList2 *Tail;
int Count;
VectorList2 *Current;
};

9.4 Matrix2

class Matrix2
{
friend Matrix3;
public:

// constructors
Matrix2();
Matrix2(const Matrix2 &x);
Matrix2(int x00,int x11);
Matrix2(int x00,int x01,int x10,int x11);
Matrix2(Vector2 Row0,Vector2 Row1);

// type converters
Matrix2(int xAll);
operator int() { return IntVal(); }
Matrix2(char *x);
operator char*() { char * rv = new char[5]; rv[0] = Data[0][0]+'0';
Matrix2(Vector2 Diagonal);
operator Vector2() { Vector2 rv; rv.Set(0,Data[0][0]);
rv.Set(1,Data[1][1]); return rv; }

// destructor
virtual ~Matrix2();

// operator overloads
Matrix2 &operator=(const Matrix2 &x);
Matrix2 &operator+=(Matrix2 &x);
Matrix2 &operator*=(Matrix2 &x);
Matrix2 operator+(Matrix2 &x);
Matrix2 operator*(Matrix2 &x);
Vector2 operator*(Vector2 &x);
int operator==(Matrix2 &x);
int operator!=(Matrix2 &x);
int operator<(Matrix2 &x);
int operator<=(Matrix2 &x);
int operator>(Matrix2 &x);
int operator>=(Matrix2 &x);
// accessors
Matrix2 * GetNext();
void SetNext(Matrix2 * NewNext);
int Get(int x, int y);
void Set(int x, int y, int NewVal);
void InsertRow(Vector2 &V, int Row);
void InsertCol(Vector2 &V, int Col);
Vector2 GetRow(int Row);
Vector2 GetCol(int Col);
// utilities
int GetEigenPoly(void);
int Determinant(void);
int Order(void);
int Singular(void);
int NonSingular(void);
Matrix2 Invert(void);
Matrix2 Transpose(void);
int Gaussian(Vector2 &Constants);
int isZero(void);
int isIdentity(void);
char *Serialize(void);
char *SerializeNL(void);
char *SerializeRow(int Row);
char *SerializeRowNL(int Row);
int IntVal(void);
int Minor(int x, int y);
int Reducible1(void);
int Reducible2(void);
int Decomposable(void);
int GetLeftComponent(void);
int GetRightComponent(void);
int isPermutation(void);
char * GetPermutation(void);
protected:
  int EigenDet(void);
  Matrix2 *Next;
  int Data[2][2];
};

9.5 Group2

class Group2
{
friend Group3;
friend Group4;
friend Group5;
public:
  // constructors
  Group2();
  Group2(Group2 &x);
  Group2(Matrix2 &x);
  Group2(Matrix2 *x);
  // destructor
  ~Group2();
  // operator overloads
Group2 &operator=(Group2 &x);
Group2 &operator+=(Group2 &x);
Group2 &operator^=(Group2 &x);
Group2 &operator*=(Group2 &x);
Group2 &operator-=(Group2 &x);
Group2 operator+(Group2 &x); // union
Group2 operator^(Group2 &x); // closed union
Group2 operator*(Group2 &x); // intersection
Group2 operator-(Group2 &x); // set difference
Group2 operator-(); // complement
int operator==(Group2 &x);
int operator!=(Group2 &x);
int operator<(Group2 &x);
int operator<=(Group2 &x);
int operator>(Group2 &x);
int operator>=(Group2 &x);
// accessors
Matrix2 * GetFirstMatrix(void);
Matrix2 * GetNextMatrix(void);
Matrix2 * GetMatrix(int x);
Group2 * GetNext(void);
void SetNext(Group2 * NewNext);
void Add(Matrix2 &x);
int Delete(Matrix2 *x);
int Order(void);
// utilites
void Close(void);
int Close(int Limit);
Group2 Conjugate(Matrix2 &x);
void Sort(void);
void Clear(void);
void GenerateGLG(void);
void GenerateAll(void);
void GenerateS2(void);
void GenerateS3(void);
Orbit2 GetOrbits(void);
Orbit2 GetVOrbits(void);
char *Serialize(void);
char *SerializeNL(void);
int Contains(Matrix2 &x);
int HasDups(void);
void RmDups(void);
void GenerateOrder(int x);
void GLGSplit(Group2 &O1,Group2 &O2,Group2 &O3);
int Reducible1(void);
int Reducible2(void);
int Decomposable(void);
int * GetLeftComponent(void);
int * GetRightComponent(void);
protected:
Group2 *Next;
Matrix2 *Head;
Matrix2 *Tail;
int Count;
Matrix2 *Current;
9.6 GroupList2

class GroupList2
{
friend GroupList3;
friend GroupList4;
friend GroupList5;
public:
    // constructors
    GroupList2();
    GroupList2(const GroupList2 &x);
    GroupList2(Group2 &x);
    GroupList2(Group2 *x);
    GroupList2(Matrix2 &x);
    GroupList2(Matrix2 *x);
    // destructor
    ~GroupList2();
    // operator overloads
    GroupList2 &operator=(const GroupList2 &x);
    GroupList2 &operator+=(const GroupList2 &x);
    GroupList2 &operator*=(GroupList2 &x);
    GroupList2 &operator-=(GroupList2 &x);
    GroupList2 operator+(const GroupList2 &x); // union
    GroupList2 operator*(GroupList2 &x); // intersection
    GroupList2 operator-(GroupList2 &x); // set difference
    GroupList2 operator-(); // complement
    int operator==(GroupList2 &x);
    int operator!=(GroupList2 &x);
    int operator<(GroupList2 &x);
    int operator<=(GroupList2 &x);
    int operator>(GroupList2 &x);
    int operator>=(GroupList2 &x);
    // accessors
    Group2 * GetFirstGroup(void);
    Group2 * GetNextGroup(void);
    Group2 * GetGroup(int x);
    GroupList2 * GetNext(void);
    void SetNext(GroupList2 * NewNext);
    void Add(Group2 &x);
    int Delete(Group2 *x);
    int Order(void);
    // utilites
    void CloseAll(void);
    void Sort(void);
    void Clear(void);
    void GenerateS2Conj(void);
    char *Serialize(void);
    char *SerializeNL(void);
    int Contains(Group2 &x);
    int HasDups(void);
    void RmDups(void);
    void GenerateAll(void);
protected:
    GroupList2 *Next;
    Group2 *Head;
    Group2 *Tail;
    int Count;
9.7 Vector3

class Vector3
{
public:

// constructors
Vector3();
Vector3(const Vector3 &x);
Vector3(int x0, int x1, int x2);

// type converters
Vector3(int xAll);
operator int() { return IntVal(); }
Vector3(char *x);
Vector3(Vector2 &x);

// destructor
~Vector3();

// operator overloads
Vector3 &operator=(const Vector3 &x);
Vector3 &operator+=(Vector3 &x);
Vector3 &operator*=(Matrix3 &x);
Vector3 &operator*=(int x);
Vector3 operator+(Vector3 &x);
int operator*(Vector3 &x);
Vector3 operator*(int x);
Vector3 operator*(Matrix3 &x);
int operator==(Vector3 &x);
int operator!=(Vector3 &x);
int operator<(Vector3 &x);
int operator<=(Vector3 &x);
int operator>(Vector3 &x);
int operator>=(Vector3 &x);

// accessors
Vector3 * GetNext();
void SetNext(Vector3 * NewNext);
Vector2 GetV2(int x);
int Get(int x);
void Set(int x, int NewVal);
int Order(void);

// utilities
int isZero(void);
int isOne(void);
char *Serialize(void);
char *SerializeNL(void);
int IntVal(void);

protected:
Vector3 *Next;
int Data[3];

};

9.8 VectorList3

class VectorList3
friend VectorList4;
friend VectorList5;
public:

// constructors
VectorList3();
VectorList3(const VectorList3 &x);
VectorList3(Vector3 &x);
VectorList3(Vector3 *x);
VectorList3(int x, int y, int z);
VectorList3(VectorList2 &x);

// destructor
~VectorList3();

// operator overloads
VectorList3 &operator=(const VectorList3 &x);
VectorList3 &operator+=(const VectorList3 &x);
VectorList3 &operator*=(VectorList3 &x);
VectorList3 &operator-=(VectorList3 &x);
VectorList3 operator+(const VectorList3 &x); // intersection
VectorList3 operator-(VectorList3 &x); // set difference
VectorList3 operator-(); // complement

// accessors
Vector3 *GetFirstVector(void);
Vector3 *GetNextVector(void);
Vector3 *GetVector(int x);
VectorList2 GetVL2(int x);
VectorList3 *GetNext(void);
void SetNext(VectorList3 *NewNext);
void Add(Vector3 &x);
void Add(int x, int y, int z);
int Delete(Vector3 *x);
int Order(void);

// utilities
void Sort(void);
void Clear(void);
char *Serialize(void);
char *SerializeNL(void);
int Contains(Vector3 &x);
int Contains(int x, int y, int z);
int HasDups(void);
void RmDups(void);
void GenerateAll(void);
void GenerateAll(int x);

protected:
VectorList3 *Next;
Vector3 *Head;
Vector3 *Tail;
int Count;
Vector3 *Current;
class Orbit3
{
friend Orbit4;
friend Orbit5;
public:
    // constructors
    Orbit3();
    Orbit3(const Orbit3 &x);
    Orbit3(VectorList3 &x);
    Orbit3(VectorList3 *x);
    Orbit3(Vector3 &x);
    Orbit3(Vector3 *x);
    Orbit3(Orbit2 &x);
    // destructor
    ~Orbit3();
    // operator overloads
    Orbit3 &operator=(const Orbit3 &x);
    Orbit3 &operator+=(const Orbit3 &x);
    Orbit3 operator+(const Orbit3 &x); // intersection
    Orbit3 operator-(Orbit3 &x); // set difference
    Orbit3 operator-(); // complement
    int operator==(Orbit3 &x);
    int operator!=(Orbit3 &x);
    int operator<(Orbit3 &x);
    int operator<=(Orbit3 &x);
    int operator>(Orbit3 &x);
    int operator>=(Orbit3 &x);
    // accessors
    VectorList3 * GetFirstList(void);
    VectorList3 * GetNextList(void);
    VectorList3 * GetList(int x);
    Orbit2 GetO2(int x);
    Orbit3 * GetNext(void);
    void SetNext(Orbit3 * NewNext);
    void Add(VectorList3 &x);
    int Delete(VectorList3 *x);
    int Order(void);
    // utilities
    void Sort(void);
    void Clear(void);
    char *Serialize(void);
    char *SerializeNL(void);
    int Contains(VectorList3 &x);
    int HasDups(void);
    void RmDups(void);
    void GenerateAll(void);
protected:
    Orbit3 *Next;
    VectorList3 *Head;
    VectorList3 *Tail;
    int Count;
    VectorList3 *Current;
};
9.10 Matrix3

class Matrix3
{
friend Matrix4;
public:
    // constructors
    Matrix3();
    Matrix3(const Matrix3 &x);
    Matrix3(int x00, int x11, int x22);
    Matrix3(int x00, int x01, int x02, int x10, int x11, int x12, int x20, int x21, int x22);
    Matrix3(Vector3 Row0, Vector3 Row1, Vector3 Row2);
    Matrix3(Matrix2 Left, int Right);
    Matrix3(int Left, Matrix2 Right);
    // type converters
    Matrix3(int xAll);
    operator int() { return IntVal(); } // Matrix3 * char *x;
    operator char*() { char *rv = new char[10];
        rv[0] = Data[0][0] + '0';
        rv[1] = Data[0][1] + '0';
        rv[2] = Data[0][2] + '0';
        rv[3] = Data[1][0] + '0';
        rv[4] = Data[1][1] + '0';
        rv[5] = Data[1][2] + '0';
        rv[6] = Data[2][0] + '0';
        rv[7] = Data[2][1] + '0';
        rv[8] = Data[2][2] + '0';
        rv[9] = '\0'; return rv; }
    Matrix3(Vector3 Diagonal);
    operator Vector3() { Vector3 rv; rv.Set(0, Data[0][0]);
        rv.Set(1, Data[1][1]);
        rv.Set(2, Data[2][2]); return rv; }
    Matrix3(Matrix2 &x);
    // destructor
    virtual ~Matrix3();
    // operator overloads
    Matrix3 &operator=(const Matrix3 &x);
    Matrix3 &operator+=(Matrix3 &x);
    Matrix3 &operator*=(Matrix3 &x);
    Vector3 operator*(Matrix3 &x);
    int operator==(Matrix3 &x);
    int operator!=(Matrix3 &x);
    int operator<(Matrix3 &x);
    int operator<=(Matrix3 &x);
    int operator>(Matrix3 &x);
    int operator>=(Matrix3 &x);
    // accessors
    Matrix3 * GetNext();
    void SetNext(Matrix3 * NewNext);
    int Get(int x, int y);
    void Set(int x, int y, int NewVal);
    Matrix2 Get2x2(int x, int y);
    void Set(int x, int y, Matrix2 NewVal);
    void InsertRow(Vector2 &V, int Row, int Col);
    void InsertCol(Vector2 &V, int Row, int Col);
    Vector2 GetRow2(int Row, int Col);
    Vector2 GetCol2(int Row, int Col);
    void InsertRow(Vector3 &V, int Row);
}
void InsertCol(Vector3 &V, int Col);
Vector3 GetRow(int Row);
Vector3 GetCol(int Col);

// utilities
int GetEigenPoly(void);
int Determinant(void);
int Order(void);
int Singular(void);
int NonSingular(void);
Matrix3 Invert(void);
Matrix3 Transpose(void);
int Gaussian(Vector3 &Constants);
int isZero(void);
int isIdentity(void);
char *Serialize(void);
char *SerializeNL(void);
char *SerializeRow(int Row);
char *SerializeRowNL(int Row);
int IntVal(void);
Matrix2 Minor(int x, int y);
int Reducible1(void);
int Reducible2(void);
int Reducible3(void);
int Reducible4(void);
int Decomposable1(void);
int Decomposable2(void);
Matrix2 GetLeftComponent(void);
Matrix2 GetRightComponent(void);
int isPermutation(void);
char * GetPermutation(void);

protected:
    int EigenDet(void);
    Matrix3 *Next;
    int Data[3][3];

};

9.11 Group3

class Group3
{
friend Group4;
friend Group5;
public:
    // constructors
    Group3();
    Group3(Group3 &x);
    Group3(Matrix3 &x);
    Group3(Matrix3 *x);
    Group3(Group2 &x);
    // destructor
    ~Group3();
    // operator overloads
    Group3 &operator=(Group3 &x);
    Group3 &operator+=(Group3 &x);
    Group3 &operator^=(Group3 &x);
    Group3 &operator*=(Group3 &x);
Group3 operator+(Group3 &x);
Group3 operator^(Group3 &x); // closed union
Group3 operator*(Group3 &x); // intersection
Group3 operator-(Group3 &x); // set difference
Group3 operator-(); // complement
int operator==(Group3 &x);
int operator!=(Group3 &x);
int operator<(Group3 &x);
int operator<=(Group3 &x);
int operator>(Group3 &x);
int operator>=(Group3 &x);
// accessors
Matrix3 * GetFirstMatrix(void);
Matrix3 * GetNextMatrix(void);
Matrix3 * GetMatrix(int x);
Group3 * GetNext(void);
void SetNext(Group3 * NewNext);
void Add(Matrix3 &x);
int Delete(Matrix3 *x);
int Order(void);
// utilities
void Close(void);
int Close(int Limit);
Group3 Conjugate(Matrix3 &x);
void Sort(void);
void Clear(void);
void GenerateGLG(void);
void GenerateAll(void);
void GenerateS2(void);
void GenerateS3(void);
void GenerateS4H(void);
void GenerateS4V(void);
Orbit3 GetOrbits(void);
Orbit3 GetVOrbits(void);
char *Serialize(void);
char *SerializeNL(void);
int Contains(Matrix3 &x);
int HasDups(void);
void RmDups(void);
void GenerateOrder(int x);
void GLGSplit(Group3 &O1, Group3 &O2, Group3 &O3, Group3 &O4, Group3 &O5, Group3 &O6, Group3 &O7);
int Reducible1(void);
int Reducible2(void);
int Reducible3(void);
int Reducible4(void);
int Decomposable1(void);
int Decomposable2(void);
Group2 GetLeftComponent(void);
Group2 GetRightComponent(void);
Group2 Get2x2(int x, int y);
protected:
Group3 *Next;
Matrix3 *Head;
Matrix3 *Tail;
int Count;
Matrix3 *Current;
9.12 GroupList3

class GroupList3
{
friend GroupList4;
friend GroupList5;
public:

    // constructors
    GroupList3();
    GroupList3(const GroupList3 &x);
    GroupList3(Group3 &x);
    GroupList3(Group3 *x);
    GroupList3(Matrix3 &x);
    GroupList3(Matrix3 *x);
    GroupList3(GroupList2 &x);

    // destructor
    ~GroupList3();

    // operator overloads
    GroupList3 &operator=(const GroupList3 &x);
    GroupList3 &operator+=(const GroupList3 &x);
    GroupList3 &operator*=(GroupList3 &x);
    GroupList3 &operator-=(GroupList3 &x);
    GroupList3 operator+(const GroupList3 &x); // intersection
    GroupList3 operator-(GroupList3 &x); // set difference
    int operator==(GroupList3 &x);
    int operator!=(GroupList3 &x);
    int operator<(GroupList3 &x);
    int operator<=(GroupList3 &x);
    int operator>(GroupList3 &x);
    int operator>=(GroupList3 &x);

    // accessors
    Group3 * GetFirstGroup(void);
    Group3 * GetNextGroup(void);
    Group3 * GetGroup(int x);
    GroupList3 * GetNext(void);
    void SetNext(GroupList3 * NewNext);
    void Add(Group3 &x);
    int Delete(Group3 *x);
    int Order(void);
    GroupList2 Get2x2(int x, int y);

    // utilities
    void CloseAll(void);
    void Sort(void);
    void Clear(void);
    void GenerateS2Conj(void);
    void GenerateS3Conj(void);
    void GenerateS4HConj(void);
    void GenerateS4VConj(void);
    char *Serialize(void);
    char *SerializeNL(void);
    int Contains(Group3 &x);
    int HasDups(void);
    void RmDups(void);
//void GenerateAll(void); // not practical, 2^512 subsets enumeration

protected:
    GroupList3 *Next;
    Group3 *Head;
    Group3 *Tail;
    int Count;
    Group3 *Current;
};

9.13 Vector4

class Vector4
{
public:

    // constructors
    Vector4();
    Vector4(const Vector4 &x);
    Vector4(int x0,int x1,int x2,int x3);
    // type converters
    Vector4(int xAll);
    operator int() { return IntVal(); }
    Vector4(char *x);
    Vector4(Vector3 &x);
    Vector4(Vector2 &x);
    // destructor
    ~Vector4();
    // operator overloads
    Vector4 &operator=(const Vector4 &x);
    Vector4 &operator+=(Vector4 &x);
    Vector4 &operator*=(Matrix4 &x);
    Vector4 &operator*=(int x);
    Vector4 operator+(Vector4 &x);
    int operator*(Vector4 &x);
    Vector4 operator*(int x);
    Vector4 operator*(Matrix4 &x);
    int operator==(Vector4 &x);
    int operator!=(Vector4 &x);
    int operator<(Vector4 &x);
    int operator<=(Vector4 &x);
    int operator>(Vector4 &x);
    int operator>=(Vector4 &x);
    // accessors
    Vector4 * GetNext();
    void SetNext(Vector4 * NewNext);
    Vector2 GetV2(int x);
    Vector3 GetV3(int x);
    int Get(int x);
    void Set(int x,int NewVal);
    // utilities
    int isZero(void);
    int isOne(void);
    char *Serialize(void);
char *SerializeNL(void);
int IntVal(void);

protected:
Vector4 *Next;
int Data[4];


9.14 VectorList4

class VectorList4
{
friend VectorList5;
public:

  // constructors
  VectorList4();
  VectorList4(const VectorList4 &x);
  VectorList4(Vector4 &x);
  VectorList4(Vector4 *x);
  VectorList4(int x, int y, int z, int w);
  VectorList4(VectorList2 &x);
  VectorList4(VectorList3 &x);

  // destructor
  ~VectorList4();

  // operator overloads
  VectorList4 &operator=(const VectorList4 &x);
  VectorList4 &operator+=(const VectorList4 &x);
  VectorList4 operator+(const VectorList4 &x); // intersection
  VectorList4 operator-(VectorList4 &x); // set difference
  VectorList4 operator-(); // complement

  int operator==(VectorList4 &x);
  int operator!=(VectorList4 &x);
  int operator<(VectorList4 &x);
  int operator<=(VectorList4 &x);
  int operator>(VectorList4 &x);
  int operator>=(VectorList4 &x);

  // accessors
  Vector4 *GetFirstVector(void);
  Vector4 *GetNextVector(void);
  Vector4 *GetVector(int x);
  VectorList2 GetVL2(int x);
  VectorList3 GetVL3(int x);
  VectorList4 *GetNext(void);
  void SetNext(VectorList4 *NewNext);
  void Add(Vector4 &x);
  void Add(int x, int y, int z, int w);
  int Delete(Vector4 *x);
  int Order(void);

  // utilities
  void Sort(void);
  void Clear(void);
  char *Serialize(void);
  char *SerializeNL(void);
  int Contains(Vector4 &x);
  int Contains(int x, int y, int z, int w);
  int HasDups(void);
  void RmDups(void);
void GenerateAll(void);
void GenerateAll(int x);

protected:
    VectorList4 *Next;
    Vector4 *Head;
    Vector4 *Tail;
    int Count;
    Vector4 *Current;
};

9.15 Orbit4

class Orbit4
{
    friend Orbit5;
public:
    // constructors
    Orbit4();
    Orbit4(const Orbit4 &x);
    Orbit4(VectorList4 &x);
    Orbit4(VectorList4 *x);
    Orbit4(Vector4 &x);
    Orbit4(Vector4 *x);
    Orbit4(Orbit2 &x);
    Orbit4(Orbit3 &x);
    // destructor
    ~Orbit4();
    // operator overloads
    Orbit4 &operator=(const Orbit4 &x);
    Orbit4 &operator+=(const Orbit4 &x);
    Orbit4 operator+(const Orbit4 &x); // intersection
    Orbit4 operator-(Orbit4 &x); // set difference
    Orbit4 operator-(); // complement
    int operator==(Orbit4 &x);
    int operator!=(Orbit4 &x);
    int operator<(Orbit4 &x);
    int operator<=(Orbit4 &x);
    int operator>(Orbit4 &x);
    int operator>=(Orbit4 &x);
    // accessors
    VectorList4 * GetFirstList(void);
    VectorList4 * GetNextList(void);
    VectorList4 * GetList(int x);
    Orbit2 Get02(int x);
    Orbit3 Get03(int x);
    Orbit4 * GetNext(void);
    void SetNext(Orbit4 * NewNext);
    void Add(VectorList4 &x);
    int Delete(VectorList4 *x);
    int Order(void);
    // utilites
    void Sort(void);
    void Clear(void);
    char *Serialize(void);
    char *SerializeNL(void);
    int Contains(VectorList4 &x);
int HasDups(void);
void RmDups(void);
void GenerateAll(void);

protected:
Orbit4 *Next;
VectorList4 *Head;
VectorList4 *Tail;
int Count;
VectorList4 *Current;

};

9.16 Matrix4

class Matrix4
{
friend Matrix5;
public:

//constructors
Matrix4();
Matrix4(const Matrix4 &x);
Matrix4(int x00,int x11,int x22, int x33);
Matrix4(int x00, int x01, int x02, int x03, int x10, int x11, int x12, int x13, int x20, int x21, int x22, int x23, int x30, int x31, int x32, int x33);
Matrix4(Vector4 Row0,Vector4 Row1,Vector4 Row3,Vector4 Row4);
Matrix4(Matrix3 Left,int Right);
Matrix4(int Left,Matrix3 Right);
Matrix4(Matrix2 Left,Matrix2 Right);
Matrix4(int Left,Matrix3 Right);
Matrix4(int Left,Matrix2 Middle,Matrix2 Right);
Matrix4(int Left,Matrix2 Right);
Matrix4(Matrix2 Left,Matrix2 Right);
Matrix4(int xAll);

// type converters
Matrix4(int xAll);
operator int() { return IntVal(); };
Matrix4(char *x);
operator char*() {char *rv = new char[17];
rv[0]=Data[0][0] + '0';rv[1]=Data[0][1] + '0';rv[2]=Data[0][2] + '0';rv[3]=Data[0][3] + '0';
rv[16]='\0';return rv; }
Matrix4(Vector4 Diagonal);
operator Vector4() {Vector4 rv; rv.Set(0,Data[0][0]);
rv.Set(1,Data[1][1]); rv.Set(2,Data[2][2]); rv.Set(3,Data[3][3]); return rv; }
Matrix4(Matrix3 &x);
Matrix4(Matrix2 &x);

//destructor
virtual ~Matrix4();

// operator overloads
Matrix4 &operator=(const Matrix4 &x);
Matrix4 &operator+=(Matrix4 &x);
Matrix4 &operator*=(Matrix4 &x);
Matrix4 operator+(Matrix4 &x);
Matrix4 operator*(Matrix4 &x);
Vector4 operator*(Vector4 &x);
int operator==(Matrix4 &x);
int operator!=(Matrix4 &x);
int operator<(Matrix4 &x);
int operator<=(Matrix4 &x);
int operator>(Matrix4 &x);
int operator>=(Matrix4 &x);

// accessors
Matrix4 * GetNext();
void SetNext(Matrix4 * NewNext);
int Get(int x, int y);
Matrix2 Get2x2(int Row, int Col);
Matrix3 Get3x3(int Row, int Col);
Vector2 GetRow2(int Row, int Col);
Vector2 GetCol2(int Row, int Col);
Vector3 GetRow3(int Row, int Col);
Vector3 GetCol3(int Row, int Col);
void Set(int Row, int Col, Matrix2 NewVal);
void Set(int Row, int Col, Matrix3 NewVal);
void InsertRow(Vector2 &V, int Row, int Col);
void InsertCol(Vector2 &V, int Row, int Col);
void InsertRow(Vector3 &V, int Row, int Col);
void InsertCol(Vector3 &V, int Row, int Col);
void InsertRow(Vector4 &V, int Row);
void InsertCol(Vector4 &V, int Col);
Vector4 GetRow(int Row);
Vector4 GetCol(int Col);

// utilities
int GetEigenPoly(void);
int Determinant(void);
int Order(void);
int Singular(void);
int NonSingular(void);
Matrix4 Invert(void);
Matrix4 Transpose(void);
int Gaussian(Vector4 &Constants);
int isZero(void);
int isIdentity(void);
char *Serialize(void);
char *SerializeNL(void);
char *SerializeRow(int Row);
char *SerializeRowNL(int Row);
int IntVal(void);
Matrix3 Minor(int x, int y);
int Reducible1(void);
int Reducible2(void);
int Reducible3(void);
int Reducible4(void);
int Reducible5(void);
int Reducible6(void);
int Decomposable1(void);
int Decomposable2(void);
int Decomposable3(void);
Matrix3 GetLeftComponent(void);
Matrix3 GetRightComponent(void);
Matrix2 GetLeft2x2(void);
Matrix2 GetRight2x2(void);
int isPermutation(void);
char * GetPermutation(void);

protected:
int EigenDet(void);
Matrix4 *Next;
int Data[4][4];

};

9.17 Group4

class Group4 {
friend Group5;
public:

// constructors
Group4();
Group4(Group4 &x);
Group4(Matrix4 &x);
Group4(Matrix4 *x);
Group4(Group3 &x);
Group4(Group2 &x);

// destructor
~Group4();

// operator overloads
Group4 &operator=(Group4 &x);
Group4 &operator+=(Group4 &x);
Group4 &operator^=(Group4 &x);
Group4 &operator*=(Group4 &x);
Group4 &operator-=(Group4 &x);
Group4 operator+(Group4 &x);
Group4 operator^(Group4 &x); // closed union
Group4 operator*(Group4 &x); // intersection
Group4 operator-(Group4 &x); // set difference
Group4 operator-(); // complement

int operator==(Group4 &x);
int operator!=(Group4 &x);
int operator<(Group4 &x);
int operator<=(Group4 &x);
int operator>(Group4 &x);
int operator>=(Group4 &x);

// accessors
Matrix4 * GetFirstMatrix(void);
Matrix4 * GetNextMatrix(void);
Matrix4 * GetMatrix(int x);
Group3 Get3x3(int x, int y);
Group2 Get2x2(int x, int y);
Group4 * GetNext(void);
void SetNext(Group4 * NewNext);
void Add(Matrix4 &x);
int Delete(Matrix4 *x);
int Order(void);
// utilites
void Close(void);
int Close(int Limit);
Group4 Conjugate(Matrix4 &x);
void Sort(void);
void Clear(void);
void GenerateGLG(void);
void GenerateAll(void);
void GenerateS2(void);
void GenerateS2A(void);
void GenerateS3(void);
void GenerateS3A(void);
void GenerateS3B(void);
void GenerateS4(void);
void GenerateS4A(void);
void GenerateS4B(void);
void GenerateS4C(void);
void GenerateS4D(void);
void GenerateS4E(void);
void GenerateS4F(void);
void GenerateS4G(void);
void GenerateS4H(void);
void GenerateS5H(void);
void GenerateS5V(void);
void GenerateS5A(void);
Orbit4 GetOrbits(void);
Orbit4 GetVOrbits(void);
char *Serialize(void);
char *SerializeNL(void);
int Contains(Matrix4 &x);
int HasDups(void);
void RmDups(void);
void GenerateOrder(int x);
void GLGSplit(Group4 &O1, Group4 &O2, Group4 &O3, Group4 &O4, Group4 &O5, Group4 &O6, Group4 &O7, Group4 &O8, Group4 &O9, Group4 &O10, Group4 &O11, Group4 &O12, Group4 &O13, Group4 &O14, Group4 &O15);
int Reducible1(void);
int Reducible2(void);
int Reducible3(void);
int Reducible4(void);
int Reducible5(void);
int Reducible6(void);
int Decomposable1(void);
int Decomposable2(void);
int Decomposable3(void);
Group3 GetLeftComponent(void);
Group3 GetRightComponent(void);
Group2 GetLeft2x2(void);
Group2 GetRight2x2(void);
protected:
Group4 *Next;
Matrix4 *Head;
Matrix4 *Tail;
int Count;
Matrix4 *Current;
}
9.18 GroupList4

class GroupList4
{
friend GroupList5;
public:

    // constructors
    GroupList4();
    GroupList4(const GroupList4 &x);
    GroupList4(const Group4 &x);
    GroupList4(const Group4 *x);
    GroupList4(const Matrix4 &x);
    GroupList4(const Matrix4 *x);
    GroupList4(const GroupList3 &x);
    GroupList4(const GroupList2 &x);

    // destructor
    ~GroupList4();

    // operator overloads
    GroupList4 &operator=(const GroupList4 &x);
    GroupList4 &operator+=(const GroupList4 &x);
    GroupList4 &operator*=(const GroupList4 &x);
    GroupList4 &operator-=(const GroupList4 &x);
    GroupList4 operator+(const GroupList4 &x);
    GroupList4 operator*(const GroupList4 &x); // intersection
    GroupList4 operator-=(const GroupList4 &x); // set difference
    GroupList4 operator-(); // complement not practical, 2^65536 subsets

    // accessors
    Group4 * GetFirstGroup(void);
    Group4 * GetNextGroup(void);
    Group4 * GetGroup(int x);
    GroupList3 Get3x3(int x, int y);
    GroupList2 Get2x2(int x, int y);
    GroupList4 * GetNext(void);
    void SetNext(GroupList4 * NewNext);
    void Add(Group4 &x);
    int Delete(Group4 *x);
    int Order(void);

    // utilites
    void CloseAll(void);
    void Sort(void);
    void Clear(void);
    void GenerateS2Conj(void);
    void GenerateS2AConj(void);
    void GenerateS3Conj(void);
    void GenerateS3AConj(void);
    void GenerateS3BConj(void);
    void GenerateS4Conj(void);
    void GenerateS4AConj(void);
    void GenerateS4BConj(void);
    void GenerateS4CConj(void);
    void GenerateS4DConj(void);
void GenerateS4EConj(void);
void GenerateS4FConj(void);
void GenerateS4GConj(void);
void GenerateS4HConj(void);
void GenerateS5HConj(void);
void GenerateS5VConj(void);
void GenerateS5AConj(void);
char *Serialize(void);
char *SerializeNL(void);
int Contains(Group4 &x);
int HasDups(void);
void RmDups(void);
//void GenerateAll(void); // not practical, 2^65536 subsets

protected:
    GroupList4 *Next;
    Group4 *Head;
    Group4 *Tail;
    int Count;
    Group4 *Current;
};

9.19 Vector5

class Vector5
{
public:
    // constructors
    Vector5();
    Vector5(const Vector5 &x);
    Vector5(int x0,int x1,int x2,int x3,int x4);
    // type converters
    Vector5(int xAll);
    operator int() { return IntVal(); }
    Vector5(char *x);
    operator char*() {char * rv= new char[6]; rv[0]=Data[0]+'0';
    rv[5]='\0'; return rv; }
    Vector5(Vector4 &x);
    Vector5(Vector3 &x);
    Vector5(Vector2 &x);
    // destructor
    ~Vector5();
    // operator overloads
    Vector5 &operator=(const Vector5 &x);
    Vector5 &operator+=(Vector5 &x);
    Vector5 &operator*=(Matrix5 &x);
    Vector5 &operator*=(int x);
    Vector5 operator+(Vector5 &x);
    int operator*(Vector5 &x);
    Vector5 operator*(int x);
    Vector5 operator*(Matrix5 &x);
    int operator==(Vector5 &x);
    int operator!=(Vector5 &x);
    int operator<(Vector5 &x);
    int operator<=(Vector5 &x);
    int operator>(Vector5 &x);
    int operator>=(Vector5 &x);
// accessors
Vector5 * GetNext();
void SetNext(Vector5 * NewNext);
int Get(int x);
void Set(int x, int NewVal);
int Order(void);
Vector2 GetV2(int x);
Vector3 GetV3(int x);
Vector4 GetV4(int x);

// utilities
int isZero(void);
int isOne(void);
char *Serialize(void);
char *SerializeNL(void);
int IntVal(void);

protected:
    Vector5 *Next;
    int Data[5];
};

9.20 VectorList5

class VectorList5
{
public:
    // constructors
    VectorList5();
    VectorList5(const VectorList5 &x);
    VectorList5(Vector5 &x);
    VectorList5(Vector5 *x);
    VectorList5(int x,int y,int z,int w,int v);
    VectorList5(VectorList2 &x);
    VectorList5(VectorList3 &x);
    VectorList5(VectorList4 &x);
    // destructor
    ~VectorList5();
    // operator overloads
    VectorList5 &operator=(const VectorList5 &x);
    VectorList5 &operator+=(const VectorList5 &x);
    VectorList5 operator+(const VectorList5 &x); // intersection
    VectorList5 operator*(VectorList5 &x); // set difference
    VectorList5 operator-(); // complement
    int operator==(VectorList5 &x);
    int operator!=(VectorList5 &x);
    int operator<(VectorList5 &x);
    int operator<=(VectorList5 &x);
    int operator>(VectorList5 &x);
    int operator>=(VectorList5 &x);
    // accessors
    Vector5 * GetFirstVector(void);
    Vector5 * GetNextVector(void);
    Vector5 * GetVector(int x);
    VectorList2 GetVL2(int x);
    VectorList3 GetVL3(int x);
    VectorList4 GetVL4(int x);
    VectorList5 * GetNext(void);
void SetNext(VectorList5 * NewNext);
void Add(Vector5 &x);
void Add(int x, int y, int z, int w, int v);
int Delete(Vector5 *x);
int Order(void);
// utilities
void Sort(void);
void Clear(void);
char *Serialize(void);
char *SerializeNL(void);
int Contains(Vector5 &x);
int Contains(int x, int y, int z, int w, int v);
int HasDups(void);
void RmDups(void);
void GenerateAll(void);
void GenerateAll(int x);
protected:
  VectorList5 *Next;
  Vector5 *Head;
  Vector5 *Tail;
  int Count;
  Vector5 *Current;
};

9.21 Orbit5

class Orbit5
{
  public:
    // constructors
    Orbit5();
    Orbit5(const Orbit5 &x);
    Orbit5(VectorList5 &x);
    Orbit5(VectorList5 *x);
    Orbit5(Vector5 &x);
    Orbit5(Vector5 *x);
    Orbit5(Orbit2 &x);
    Orbit5(Orbit3 &x);
    Orbit5(Orbit4 &x);
    // destructor
    ~Orbit5();
    // operator overloads
    Orbit5 &operator=(const Orbit5 &x);
    Orbit5 &operator+=(const Orbit5 &x);
    Orbit5 operator+(const Orbit5 &x); // intersection
    Orbit5 operator-(Orbit5 &x); // set difference
    //Orbit5 operator-(); // complement not practical 4 billion sets
    int operator==(Orbit5 &x);
    int operator!=(Orbit5 &x);
    int operator<(Orbit5 &x);
    int operator<=(Orbit5 &x);
    int operator>(Orbit5 &x);
    int operator>=(Orbit5 &x);
    // accessors
    VectorList5 * GetFirstList(void);
    VectorList5 * GetNextList(void);
VectorList5 * GetList(int x);
Orbit2 GetO2(int x);
Orbit3 GetO3(int x);
Orbit4 GetO4(int x);
Orbit5 * GetNext(void);
void SetNext(Orbit5 * NewNext);
void Add(VectorList5 &x);
int Delete(VectorList5 *x);
int Order(void);
// utilities
void Sort(void);
void Clear(void);
char *Serialize(void);
char *SerializeNL(void);
int Contains(VectorList5 &x);
int HasDups(void);
void RmDups(void);
//void GenerateAll(void); // Not Practical 4 billion sets
protected:
Orbit5 *Next;
VectorList5 *Head;
VectorList5 *Tail;
int Count;
VectorList5 *Current;
};

9.22 Matrix5

class Matrix5
{
public:

    // constructors
    Matrix5();
    Matrix5(const Matrix5 &x);
    Matrix5(int x00,int x11,int x22,int x33,int x44);
    Matrix5(int x00,int x01,int x02,int x03,int x04,int x10,int x11,int x12,int x13,int x14,int x20,int x21,int x22,int x23,int x24,int x30,int x31,int x32,int x33,int x34,int x40,int x41,int x42,int x43,int x44);
    Matrix5(Vector5 Row0,Vector5 Row1,Vector5 Row2,Vector5 Row3,Vector5 Row4);
    Matrix5(Matrix4 Left,int Right);
    Matrix5(int Left,Matrix4 Right);
    Matrix5(Matrix3 Left,Matrix2 Right);
    Matrix5(Matrix2 Left,Matrix3 Right);
    Matrix5(Matrix3 Left,int Middle,Matrix2 Right);
    Matrix5(int Left,Matrix3 Middle,Matrix2 Right);
    Matrix5(int Left,int Middle,Matrix3 Right);
    Matrix5(Matrix2 Left,Matrix2 Middle,int Right);
    Matrix5(int Left,int Middle,Matrix2 Middle,Matrix2 Right);
    Matrix5(int xAll);
    operator int() { return IntVal(); }
Matrix5(char *x);
operator char*() {char * rv= new char[26];

rv[25]='\0'; return rv; }

Matrix5(Vector5 Diagonal);
operator Vector5() {Vector5 rv; rv.Set(0,Data[0][0]); rv.Set(1,Data[1][1]); rv.Set(2,Data[2][2]); rv.Set(3,Data[3][3]); rv.Set(4,Data[4][4]); return rv; }

Matrix5 &operator=(const Matrix5 &x);
Matrix5 &operator+=(Matrix5 &x);
Matrix5 &operator*=(Matrix5 &x);
Matrix5 operator+(Matrix5 &x);
Matrix5 operator*(Matrix5 &x);
Vector5 operator*(Vector5 &x);

int operator==(Matrix5 &x);
int operator!=(Matrix5 &x);
int operator<(Matrix5 &x);
int operator<=(Matrix5 &x);
int operator>(Matrix5 &x);
int operator>=(Matrix5 &x);

// accessors
Matrix5 * GetNext();
void SetNext(Matrix5 * NewNext);
int Get(int x, int y);
Matrix2 Get2x2(int Row, int Col);
Matrix3 Get3x3(int Row, int Col);
Matrix4 Get4x4(int Row, int Col);
void Set(int Row, int Col, Matrix2 NewVal);
void Set(int Row, int Col, Matrix3 NewVal);
void Set(int Row, int Col, Matrix4 NewVal);
Vector2 GetRow2(int Row, int Col);
Vector2 GetCol2(int Row, int Col);
Vector3 GetRow3(int Row, int Col);
Vector3 GetCol3(int Row, int Col);
Vector4 GetRow4(int Row, int Col);
Vector4 GetCol4(int Row, int Col);
void InsertRow(Vector2 &V, int Row, int Col);
void InsertCol(Vector2 &V, int Row, int Col);
void InsertRow(Vector3 &V, int Row, int Col);
void InsertCol(Vector3 &V, int Row, int Col);
void InsertRow(Vector4 &V, int Row, int Col);
void InsertCol(Vector4 &V, int Row, int Col);
void Set(int x, int y, int NewVal);
void InsertRow(Vector5 &V, int Row);
void InsertCol(Vector5 &V, int Col);
Vector5 GetRow(int Row);
Vector5 GetCol(int Col);

// utilities
int GetEigenPoly(void);
int Determinant(void);
int Order(void);
int Singular(void);
int NonSingular(void);
Matrix5 Invert(void);
Matrix5 Transpose(void);
int Gaussian(Vector5 &Constants);
int isZero(void);
int isIdentity(void);
char *Serialize(void);
char *SerializeNL(void);
char *SerializeRow(int Row);
char *SerializeRowNL(int Row);
int IntVal(void);
Matrix4 Minor(int x, int y);
int isPermutation(void);
int Reducible1(void);
int Reducible2(void);
int Reducible3(void);
int Reducible4(void);
int Reducible5(void);
int Reducible6(void);
int Reducible7(void);
int Reducible8(void);
int Decomposable1(void);
int Decomposable2(void);
int Decomposable3(void);
int Decomposable4(void);
Matrix4 GetLeftComponent(void);
Matrix4 GetRightComponent(void);
Matrix3 GetLeft3x3(void);
Matrix3 GetRight3x3(void);
Matrix2 GetLeft2x2(void);
Matrix2 GetRight2x2(void);
char * GetPermutation(void);

protected:
    int EigenDet(void);
Matrix5 *Next;
int Data[5][5];

};
public:
    // constructors
    Group5();
    Group5(Group5 &x);
    Group5(Matrix5 &x);
    Group5(Matrix5 *x);
    Group5(Group2 &x);
    Group5(Group3 &x);
    Group5(Group4 &x);
    // destructor
    ~Group5();
    // operator overloads
    Group5 &operator=(Group5 &x);
    Group5 &operator+=(Group5 &x);
    Group5 &operator^=(Group5 &x);
    Group5 &operator*=(Group5 &x);
    Group5 &operator-=(Group5 &x);
    Group5 operator+(Group5 &x);
    Group5 operator^(Group5 &x); // closed union
    Group5 operator*(Group5 &x); // intersection
    Group5 operator-(Group5 &x); // set difference
    Group5 operator-(); // complement//never tested
    int operator==(Group5 &x);
    int operator!=(Group5 &x);
    int operator<(Group5 &x);
    int operator<=(Group5 &x);
    int operator>(Group5 &x);
    int operator>=(Group5 &x);
    // accessors
    Matrix5 * GetFirstMatrix(void);
    Matrix5 * GetNextMatrix(void);
    Matrix5 * GetMatrix(int x);
    Group2 Get2x2(int x, int y);
    Group3 Get3x3(int x, int y);
    Group4 Get4x4(int x, int y);
    Group5 * GetNext(void);
    void SetNext(Group5 * NewNext);
    void Add(Matrix5 &x);
    int Delete(Matrix5 *x);
    int Order(void);
    // utilities
    void Close(void);
    int Close(int Limit);
    Group5 Conjugate(Matrix5 &x);
    void Sort(void);
    void Clear(void);
    void GenerateGLG(void);
    void GenerateAll(void); // Requires 64bit OS
    void GenerateS2(void);
    void GenerateS3(void);
    void GenerateS4(void);
    void GenerateS5(void);
    void GenerateS5A(void);
    void GenerateS5B(void);
    void GenerateS5C(void);
    void GenerateS5D(void);
    void GenerateS6H(void);
void GenerateS6V(void);
Orbit5 GetOrbits(void);
Orbit5 GetVOrbits(void);
char *Serialize(void);
char *SerializeNL(void);
int Contains(Matrix5 &x);
int HasDups(void);
void RmDups(void);
void GenerateOrder(int x);
void GLGSplit(Group5 &O1, Group5 &O2, Group5 &O3, Group5 &O4, Group5 &O5, Group5 &O6, Group5 &O7, Group5 &O8, Group5 &O9, Group5 &O10, Group5 &O11, Group5 &O12, Group5 &O13, Group5 &O14, Group5 &O15, Group5 &O16, Group5 &O17, Group5 &O18, Group5 &O19, Group5 &O20, Group5 &O21, Group5 &O22, Group5 &O23, Group5 &O24, Group5 &O25, Group5 &O26, Group5 &O27, Group5 &O28, Group5 &O29, Group5 &O30, Group5 &O31);
);//requires 64 bit OS
int Reducible1(void);
int Reducible2(void);
int Reducible3(void);
int Reducible4(void);
int Reducible5(void);
int Reducible6(void);
int Reducible7(void);
int Reducible8(void);
int Decomposable1(void);
int Decomposable2(void);
int Decomposable3(void);
int Decomposable4(void);
Group4 GetLeftComponent(void);
Group4 GetRightComponent(void);
Group3 GetLeft3x3(void);
Group3 GetRight3x3(void);
Group2 GetLeft2x2(void);
Group2 GetRight2x2(void);
protected:
Group5 *Next;
Matrix5 *Head;
Matrix5 *Tail;
int Count;
Matrix5 *Current;
};

9.24 GroupList5

class GroupList5
{
public:

    // constructors
    GroupList5();
    GroupList5(const GroupList5 &x);
    GroupList5(Group5 &x);
    GroupList5(Group5 *x);
    GroupList5(Matrix5 &x);
    GroupList5(Matrix5 *x);
    GroupList5(GroupList2 &x);
}
GroupList5(GroupList3 &x);
GroupList5(GroupList4 &x);
// destructor
~GroupList5();
// operator overloads
GroupList5 &operator=(const GroupList5 &x);
GroupList5 &operator+=(const GroupList5 &x);
GroupList5 &operator*=(GroupList5 &x);
GroupList5 &operator-=(GroupList5 &x);
GroupList5 operator+(const GroupList5 &x);  // intersection
GroupList5 operator-(GroupList5 &x);  // set difference
//GroupList5 operator-();  // complement not practical, 2^33554432
subsets
  int operator==(GroupList5 &x);
  int operator!=(GroupList5 &x);
  int operator<(GroupList5 &x);
  int operator<=(GroupList5 &x);
  int operator>(GroupList5 &x);
  int operator>=(GroupList5 &x);
// accessors
  Group5 * GetFirstGroup(void);
  Group5 * GetNextGroup(void);
  Group5 * GetGroup(int x);
  GroupList2 Get2x2(int x, int y);
  GroupList3 Get3x3(int x, int y);
  GroupList4 Get4x4(int x, int y);
  GroupList5 * GetNext(void);
  void SetNext(GroupList5 * NewNext);
  void Add(Group5 &x);
  int Delete(Group5 *x);
  int Order(void);
// utilites
  void CloseAll(void);
  void Sort(void);
  void Clear(void);
  void GenerateS2Conj(void);
  void GenerateS3Conj(void);
  void GenerateS4Conj(void);
  void GenerateS5Conj(void);
  void GenerateS5AConj(void);
  void GenerateS5BConj(void);
  void GenerateS5CConj(void);
  void GenerateS5DConj(void);
  char *Serialize(void);
  char *SerializeNL(void);
  int Contains(Group5 &x);
  int HasDups(void);
  void RmDups(void);
//void GenerateAll(void);  // not practical, 2^33554432 subsets
protected:
  GroupList5 *Next;
  Group5 *Head;
  Group5 *Tail;
  int Count;
  Group5 *Current;
9.25 GenerateGLG2

class GenerateGLG2: public Matrix2
{
public:
    // constructors
    GenerateGLG2();
    GenerateGLG2(const GenerateGLG2 &x);
    // type converter
    GenerateGLG2(Matrix2 &x);
    // destructor
    ~GenerateGLG2();
    // operator overloads
    GenerateGLG2 &operator=(const GenerateGLG2 &x);
    // accessors
    // utilities
    int First();
    int Next();
    int First(int x);
    int Next(int x);

protected:
    int InUse[4];
    int RowValue[2];
};

9.26 GenerateAll2

class GenerateAll2: public Matrix2
{
public:
    // constructors
    GenerateAll2();
    GenerateAll2(const GenerateAll2 &x);
    // type converter
    GenerateAll2(Matrix2 &x);
    // destructor
    ~GenerateAll2();
    // operator overloads
    GenerateAll2 &operator=(const GenerateAll2 &x);
    // accessors
    // utilities
    int First();
    int Next();
    int First(int x);
    int Next(int x);

protected:
    int RowValue[2];
};

9.27 GenerateGLG3

class GenerateGLG3: public Matrix3
{
public:
    // constructors
    GenerateGLG3();
    GenerateGLG3(const GenerateGLG3 &x);
    // utility
    GenerateGLG3(Matrix3 &x);
    ~GenerateGLG3();
    // operator overloads
    GenerateGLG3 &operator=(const GenerateGLG3 &x);
    // accessors
    // utilities
    int First();
    int Next();
    int First(int x);
    int Next(int x);

protected:
    int InUse[4];
    int RowValue[2];
};
9.28 GenerateAll3

class GenerateAll3: public Matrix3
{
public:
    //constructors
    GenerateAll3();
    GenerateAll3(const GenerateAll3 &x);
    //type converter
    GenerateAll3(const GenerateAll3 &x);
    //destructor
    ~GenerateAll3();
    //operator overloads
    GenerateAll3 &operator=(const GenerateAll3 &x);
    //accessors
    //utilities
    int First();
    int Next();
    int First(int x);
    int Next(int x);
protected:
    int InUse[8];
    int RowValue[3];
};

9.29 GenerateGLG4

class GenerateGLG4: public Matrix4
{
public:
    //constructors
    GenerateGLG4();
    GenerateGLG4(const GenerateGLG4 &x);
    //type converter
    GenerateGLG4(const GenerateGLG4 &x);
    //destructor
    ~GenerateGLG4();
    //operator overloads
    GenerateGLG4 &operator=(const GenerateGLG4 &x);
9.30 GenerateAll4

class GenerateAll4: public Matrix4
{
public:
  //constructors
  GenerateAll4();
  GenerateAll4(const GenerateAll4 &x);
  //type converter
  GenerateAll4(Matrix4 &x);
  //destructor
  ~GenerateAll4();
  //operator overloads
  GenerateAll4 &operator=(const GenerateAll4 &x);
  //accessors
  //utilities
  int First();
  int Next();
  int First(int x);
  int Next(int x);
protected:
  int RowValue[4];
};

9.31 GenerateGLG5

class GenerateGLG5: public Matrix5
{
public:
  //constructors
  GenerateGLG5();
  GenerateGLG5(const GenerateGLG5 &x);
  //type converter
  GenerateGLG5(Matrix5 &x);
  //destructor
  ~GenerateGLG5();
  //operator overloads
  GenerateGLG5 &operator=(const GenerateGLG5 &x);
  //accessors
  //utilities
  int First();
  int Next();
  int First(int x);
  int Next(int x);
protected:
  int InUse[32];
9.32 GenerateAll5

class GenerateAll5: public Matrix5
{
public:

    // constructors
    GenerateAll5();
    GenerateAll5(const GenerateAll5 &x);

    // type converter
    GenerateAll5(Matrix5 &x);

    // destructor
    ~GenerateAll5();

    // operator overloads
    GenerateAll5 &operator=(const GenerateAll5 &x);

    // accessor
    int First();
    int Next();
    int First(int x);
    int Next(int x);

protected:
    int RowValue[5];
};