User's Manual* for the Expanded Coordinate-Free Geometric Programming System

William J.R. Longabaugh

March 13, 1992

1 Lists

Lists (actually variable-sized arrays) are provided as the primary way to pass a variable number of objects to and from functions. Because the semantics of lists are that they store and return actual copies of the list elements, instead of using pointers to elements, they are *not* implemented using inheritance from a base class.

1.1 ScalarList

Constructors

`ScalarList(int n)`

Builds a list of length n, with each entry initialized to 0.0.

`ScalarList(Scalar s, int n)`

Builds a list of length n, with each entry initialized to s. CAUTION: Note that it is very easy to confuse this constructor with the one that takes two scalars as arguments.

```
ScalarList(Scalar s)
ScalarList(Scalar s1, Scalar s2)
ScalarList(Scalar s1, Scalar s2, Scalar s3)
ScalarList(Scalar s1, Scalar s2, Scalar s3, Scalar s4)
```

Each builds a list and initializes it with the specified values.

*This "manual" only gives the briefest description for functions in the system. For more complete treatment, consult William J. R. Longabaugh, "An Expanded System for Coordinate-Free Geometric Programming", master's thesis, University of Washington, 1992.

Interrogation and Access

`int Length(void)`

Returns the length of the list.

`Scalar& operator[](int n)`

Returns a reference to the n^{th} scalar in the list (using zero-based indexing).

Concatenation and Duplication

`ScalarList operator+(ScalarList& l)`

Concatenates the list l on the end of this list and returns the resulting list.

`ScalarList operator*(int i)`

Duplicates this list i times, concatenates the copies, and returns the resulting list.

Miscellaneous

```
ScalarList(void)
ScalarList(ScalarList& t)
ScalarList& operator=(ScalarList& t)
~ScalarList()
void debug_out(ostream& c, int indent)
```

Standard housekeeping functions.

1.2 IntList

Constructors

`IntList(int listsize)`

Builds a list of length *listsize*, with each entry initialized to 0.

`IntList(int element, int listsize)`

Builds a list of length *listsize*, with each entry initialized to *element*.

```
IntList(int element1, int element2, int element3)
IntList(int element1, int element2, int element3, int element4)
```

Note that some shortcut constructors cannot be implemented due to ambiguity with the previously defined constructors. The given constructors each builds a list and initializes it with the specified values.

Interrogation and Access

`int Length(void)`

Returns the length of the list.

`int& operator[](int n)`

Returns a reference to the n^{th} integer in the list (using zero-based indexing).

Concatenation and Duplication

`IntList operator+(IntList& l)`

Concatenates the list l on the end of this list and returns the resulting list.

`IntList operator*(int i)`

Duplicates this list i times, concatenates the copies, and returns the resulting list.

Miscellaneous

```
IntList(void)
IntList(IntList& t)
IntList& operator=(IntList& t)
~IntList()
void debug_out(ostream& c, int indent)
```

Standard housekeeping functions.

1.3 GeObList

Constructors

`GeObList(int n)`

Builds a list of length n, with each entry initialized to NULL_GEOB.

`GeObList(GeOb& g, int n)`

Builds a list of length n, with each entry initialized to g.

```
GeObList(GeOb& m)
GeObList(GeOb& m1, GeOb& m2)
GeObList(GeOb& m1, GeOb& m2, GeOb& m3)
GeObList(GeOb& m1, GeOb& m2, GeOb& m3, GeOb& m4)
```

Each builds a list and initializes it with the specified objects.

Interrogation and Access

`int Length(void)`

Returns the length of the list.

`GeOb& operator[](int n)`

Returns the n^{th} GeOb in the list (using zero-based indexing).

Concatenation and Duplication

`GeObList operator+(GeObList& g)`

Concatenates the list g on the end of this list and returns the resulting list.

`GeObList operator*(int s)`

Duplicates this list s times, concatenates the copies, and returns the resulting list.

Miscellaneous

```
GeObList(void)
GeObList(GeObList& t)
GeObList& operator=(GeObList& t)
~GeObList()
void debug_out(ostream& c, int indent)
```

Standard housekeeping functions.

1.4 SpaceList

Constructors

`SpaceList(int n)`

Builds a list of length n, with each entry initialized to NULL_SPACE.

`SpaceList(Space& g, int n)`

Builds a list of length n, with each entry initialized to g.

```
SpaceList(Space& m)
SpaceList(Space& m1, Space& m2)
SpaceList(Space& m1, Space& m2, Space& m3)
SpaceList(Space& m1, Space& m2, Space& m3, Space& m4)
```

Each builds a list and initializes it with the specified spaces.

Interrogation and Access

`int Length(void)`

Returns the length of the list.

`Space& operator[](int n)`

Returns a reference to the n^{th} space in the list (using zero-based indexing).

Concatenation and Duplication

`SpaceList operator+(SpaceList& g)`

Concatenates the list g on the end of this list and returns the resulting list.

`SpaceList operator*(int s)`

Duplicates this list s times, concatenates the copies, and returns the resulting list.

Miscellaneous

```
SpaceList(void)
SpaceList(SpaceList& t)
SpaceList& operator=(SpaceList& t)
~SpaceList()
void debug_out(ostream& c, int indent)
```

Standard housekeeping functions.

1.5 BasisList

Constructors

`BasisList(int n)`

Builds a list of length n, with each entry initialized to NULL_BASIS.

`BasisList(Basis& g, int n)`

Builds a list of length n, with each entry initialized to g.

```
BasisList(Basis& m)
BasisList(Basis& m1, Basis& m2)
BasisList(Basis& m1, Basis& m2, Basis& m3)
BasisList(Basis& m1, Basis& m2, Basis& m3, Basis& m4)
```

Each builds a list and initializes it with the specified bases.

Interrogation and Access

`int Length(void)`

Returns the length of the list.

`Basis& operator[](int n)`

Returns the n^{th} basis in the list (using zero-based indexing).

Concatenation and Duplication

`BasisList operator+(BasisList& g)`

Concatenates the list g on the end of this list and returns the resulting list.

`BasisList operator*(int s)`

Duplicates this list s times, concatenates the copies, and returns the resulting list.

Miscellaneous

```
BasisList(void)
BasisList(BasisList& t)
BasisList& operator=(BasisList& t)
~BasisList()
void debug_out(ostream& c, int indent)
```

Standard housekeeping functions.

2 Spaces

2.1 Member Functions for the Base Class Space

Interrogation Functions

`int Dim(void)`

Returns the dimension of the space.

`char* Name(char* buf)`

Fills in the specified buffer with the debug name of the space, and returns a pointer to the buffer.

`SpaceType Holds(void)`

Returns the type of space currently held by the space.

`Basis StdBasis(void)`

Returns the "standard basis" for the space.

`SRel SpaceRelation(Space& s)`

Returns a tag that says how space s is related to this space.

`SRel ThisSpaceIs(void)`

Returns a tag that gives the position of this space in the six-space set.

`Boolean HasSpace(SRel s)`

Returns TRUE if there exists a space in the six-space set that sits in relation s to this space. Since spaces are created lazily, the function returns FALSE if the space has not been created yet.

`Space GetSpace(SRel s)`

Returns the space that sits in relation s to this space.

`AffineMap AffineMapTo(SRel s)`

Returns the affine map to the space that sits in relation s to this space, if it exists. Only valid if this space is an affine, linearization, or projective completion space.

`ProjectiveMap ProjectiveMapTo(SRel s)`

Returns the projective map to the space that sits in relation s to this space, if it exists. Not valid if this space is not a linearization or projective completion space.

`LinearMap LinearMapTo(SRel s)`

Returns the linear map to the space that sits in relation s to this space, if it exists. Not valid if this space is not a linearization or tangent space.

`VSpace Dual(void)`

Returns the dual space to this space. Only valid if this space is a vector space.

`Simplex StdSimplex(void)`

Returns the "standard simplex" for this space. Only valid if this space is an affine space.

`Boolean IsEuclidean(void)`

Returns TRUE if this space is a Euclidean space, FALSE otherwise.

`MLM InnerProduct(void)`

Returns the inner product for this space. Only valid if this space is a Euclidean vector space.

`MLM CrossProduct(void)`

Returns the cross product for this space. Only valid if this space is a Euclidean vector space.

`GeObList PtsAtInfinity(void)`

Returns a list of points that span the set of points at infinity. The list will be empty unless this space is a projective space with a standard affine subset.

`SubSet FullSet(void)`

Returns a subset that contains all objects in the space.

`PSubSet FullProjSet(void)`

Returns a projective subset containing all the projective points or vector equivalence classes in a space. Not valid for affine spaces.

`ASubSet StdAffineSubset(void)`

Returns the standard affine subset for a space.

`GeObType NativeType(void)`

Returns a tag indicating what the native type of object is for a space (e.g. Vectors for a vector space that is not a tangent space, AVectors for tangent spaces, PPoints for projective spaces).

Comparison Operators

`Boolean operator==(Space& s)`

Returns TRUE if and only if the space s is the same as this space. Two distinct spaces with the same cartesian product structure are *not* considered equal.

`Boolean operator!=(Space& s)`

Returns TRUE if and only if the space s is not the same as this space.

Cartesian Product Operations

`Space operator[](int n)`

Returns the n^{th} component space of a cartesian product space. Indexing is zero-based.

`int CPSpaceSize(void)`

Returns the number of component spaces in a cartesian product space. Simple spaces return 1.

Manual Stitching Operations

`Space SetSpace(Space& s, Map& m)`

Used to manually stitch together spaces into a six-space set. Space s is stitched to this space using map m, which must be a map from this space to s.

Miscellaneous

```
Space(void)
Space(Space& v)
Space& operator=(Space& s)
void debug_out(ostream& c, int indent)
void heavy_debug_out(ostream& c, int indent)
```

Standard housekeeping functions. The function *debug_out*() does not print out member objects that themselves contain Space member objects (which avoids infinite recursion).

2.2 Member Functions for VSpace

Constructors

`VSpace(Space& s)`

Used to downcast a general space to a vector space. Only succeeds if the general space is holding a vector space.

`VSpace(char* namein, int n, Boolean eflag)`

Builds a vector space of dimension n, with debug name in buffer *namein*. The space creates its own copy of the name. If the flag $eflag$ is TRUE, the space will be Euclidean.

`VSpace(char* namein, Boolean eflag, ASpace& a, PSpace& p)`

Builds a vector space, linking it to the spaces a and p to create the core of a six-space set. The space creates its own copy of the debug name. This space is Euclidean if $eflag$ is TRUE.

```
VSpace(char* namein, Boolean eflag, Space& ins)
```

Builds a vector space, linking it to the space ins, which is either an affine or projective space. The space creates its own copy of the debug name. This space is Euclidean if $eflag$ is TRUE.

```
VSpace(char* namein, SpaceList& t, Boolean eflag)
```

Builds a cartesian product vector space, using the spaces in the list t as component spaces. The space creates its own copy of the debug name. If the flag $eflag$ is TRUE, the space will be Euclidean.

Manual Stitching Operations

```
Space SetSpace(Space& s, Map& m)
```

See the description given in Section **??** for the base class (this is implemented as a virtual function).

Miscellaneous

```
VSpace(void)
VSpace(VSpace& v)
VSpace& operator=(VSpace& s)
void debug_out(ostream& c, int indent)
```

Standard housekeeping functions.

2.3 Member Functions for ASpace

Constructors

```
ASpace(Space& s)
```

Used to downcast a general space to an affine space. Only succeeds if the general space is holding an affine space.

```
ASpace(char* namein, int n, Boolean eflag)
```

Builds an affine space of dimension n, with debug name in buffer $namein$. The space creates its own copy of the name. If $eflag$ is TRUE, the space will be Euclidean.

```
ASpace(char* namein, Boolean eflag, VSpace& v, PSpace& p)
```

Builds an affine space, linking it to the spaces v and p to create the core of a six-space set. The space creates its own copy of the debug name. This space is Euclidean if $eflag$ is TRUE.

```
ASpace(char* namein, Boolean eflag, Space& ins)
```

Builds an affine space, linking it to the space ins, which is either a vector or projective space. The space creates its own copy of the debug name. This space is Euclidean if $eflag$ is TRUE.

```
ASpace(char* namein, SpaceList& t, Boolean eflag)
```

Builds a cartesian product affine space, using the spaces in the list t as component spaces. The space creates its own copy of the debug name. If the flag $eflag$ is TRUE, the space will be Euclidean.

Manual Stitching Operations

```
Space SetSpace(Space& s, Map& m)
```

See the description given in Section **??** for the base class (this is implemented as a virtual function).

Miscellaneous

```
ASpace(void)
ASpace(ASpace& s)
ASpace& operator=(ASpace& s)
void debug_out(ostream& c, int indent)
```

Standard housekeeping functions.

2.4 Member Functions for PSpace

Constructors

```
PSpace(Space& s)
```

Used to downcast a general space to an affine space. Only succeeds if the general space is holding a projective space.

```
PSpace(char* namein, int n)
```

Builds a projective space of dimension n, with debug name in buffer $namein$. The space creates its own copy of the name.

```
PSpace(char* namein, VSpace& v, ASpace& a)
```

Builds a projective space, linking it to the spaces v and a to create the core of a six-space set. The space creates its own copy of the debug name.

`PSpace(char* namein, Space& v)`

Builds a projective space, linking it to the space v, which is either a vector or affine space. The space creates its own copy of the debug name.

Manual Stitching Operations

`Space SetSpace(Space& s, Map& m)`

See the description given in Section **??** for the base class (this is implemented as a virtual function).

Miscellaneous

```
PSpace(void)
PSpace(PSpace& v)
PSpace& operator=(PSpace& s)
void debug_out(ostream& c, int indent)
```

Standard housekeeping functions.

3 SubSet

3.1 Member Functions for the Base Class SubSet

Interrogation Functions

`char* Name(char* buf)`

Fills in the specified buffer with the debug name of the subset, and returns a pointer to the buffer.

`int Dim(void)`

Returns the dimension of the subset. If the subset is projective with removed points, this dimension reflects the dimension of the related primitive geometric form (e.g. a projective plane with a removed point has a dimension of one).

`SubSetType Holds(void)`

Returns the type of subset currently held by the subset.

`Boolean IsIn(GeOb& g)`

Returns TRUE if and only if the geometric object g is in the subset.

`Space EmbeddingSpace(void)`

Returns the space that contains the subset.

`Boolean IsSubset(SubSet& s)`

Returns TRUE if and only if the subset s is a subset of this subset. Not valid for projective subsets with removed points. Will return FALSE if the subset types do not match.

`Boolean IsFullSpace(void)`

Returns TRUE if and only if this subset contains all the geometric objects in the embedding space.

`Boolean HasRemovedPoints(void)`

Only valid if the subset is a projective subset. Returns TRUE if and only if the subset has a set of removed points (i.e. it is the domain of a noninvertible projective map).

`GeObList AtInfinity(void)`

Returns a list of points that span the set of points at infinity. Only valid if the subset is an affine subset in a projective space.

`VSubSet TangentSub(void)`

Returns the subset of associated tangent vectors. Only valid if the subset is an affine subset in an affine or vector space.

`GeObType Accepts(void)`

Returns the type of geometric object that is contained in the subset (e.g. affine vector equivalence classes in a projective subset defined in a tangent space).

Miscellaneous

```
SubSet(void)
SubSet(SubSet& s)
SubSet& operator=(SubSet& s)
void debug_out(ostream& c, int indent)
```

Standard housekeeping functions.

3.2 Member Functions for VSubSet

Constructors

```
VSubSet(char* namein, VSpace& s, GeObList& v)
```

Builds a vector subset of vector space s, with debug name in buffer *namein*. The subset creates its own copy of the name. The geometric objects in list v, after mapping into space s if necessary, span the subset.

```
VSubSet(SubSet& s)
```

Used to downcast a general subset to a vector subset. Only succeeds if the general subset is holding a vector subset.

Miscellaneous

```
VSubSet(void)
VSubSet(VSubSet& s)
VSubSet& operator=(VSubSet& s)
void debug_out(ostream& c, int indent)
```

Standard housekeeping functions.

3.3 Member Functions for ASubSet

Constructors

```
ASubSet(char* namein, Space& s, GeObList& v)
```

Builds an affine subset of vector or affine space s, with debug name in buffer *namein*. The subset creates its own copy of the name. The geometric objects in list v, after mapping into space s if necessary, span the subset.

```
ASubSet(char* namein, PSpace& s, GeOb& v, GeObList& inf)
```

Builds an affine subset of projective space s, with debug name in buffer *namein*. The subset creates its own copy of the name. The geometric objects v and in list inf, after mapping into space s if necessary, define a projective subspace. The subsequent subtraction of the points at infinity specified by list inf creates an affine subset.

```
ASubSet(SubSet& s)
```

Used to downcast a general subset to an affine subset. Only succeeds if the general subset is holding an affine subset.

Miscellaneous

```
ASubSet(void)
ASubSet(ASubSet& s)
ASubSet& operator=(ASubSet& s)
void debug_out(ostream& c, int indent)
```

Standard housekeeping functions.

3.4 Member Functions for PSubSet

Constructors

```
PSubSet(char* namein, Space& s, GeObList& v)
```

Builds a projective subset of space s, with debug name in buffer *namein*. The subset creates its own copy of the name. The geometric objects in list v, after mapping into space s if necessary, span the subset.

```
PSubSet(char* namein, Space& s, GeObList& bp, GeObList& v)
```

Builds a projective subset of space s, with debug name in buffer *namein*. The subset creates its own copy of the name. The geometric objects in lists bp and v, after mapping into space s if necessary, span some projective subspace. The objects in bp are base points that span the set of removed points that are extracted from the subset. This constructor is used to build domain subsets for noninvertible projective maps.

```
PSubSet(SubSet& s)
```

Used to downcast a general subset to a projective subset. Only succeeds if the general subset is holding a projective subset.

Miscellaneous

```
PSubSet(void)
PSubSet(PSubSet& s)
PSubSet& operator=(PSubSet& s)
void debug_out(ostream& c, int indent)
```

Standard housekeeping functions.

4 Basis

4.1 Member Functions for the Base Class Basis

Interrogation Functions

`Space SpaceOf(void)`

Returns the space that contains the basis.

`BasisType Holds(void)`

Returns the type of basis currently held by the basis.

`char* Name(char* buf)`

Fills in the specified buffer with the debug name of the basis, and returns a pointer to the buffer.

`GeOb operator[](int n)`

Returns the n^{th} object from the basis, using zero-based indexing. For k-dimensional frames, the point is the k^{th} object.

`ScalarList operator()(GeOb& v)`

Returns the coordinates of the object v with respect to this basis. The object is first mapped into the space of the basis, if necessary.

`VBasis Dual(void)`

Returns the vector basis that is the dual of this basis. Only valid for vector bases.

Miscellaneous

```
Basis(void)
Basis(Basis& b)
Basis& operator=(Basis& b)
void debug_out(ostream& c, int indent)
```

Standard housekeeping functions.

4.2 Member Functions for Simplex

Constructors

`Simplex(char* namein, GeOb& t)`

If t is a point tuple from a cartesian product affine space $A \times \ldots \times A$, this builds a simplex for A from the tuple members. The simplex creates its own copy of the debug name given in the buffer *namein*.

```
Simplex(char* namein, ASpace& ins, GeObList& t)
```

Builds a simplex for the affine space *ins*, using the objects in the list t, after they are mapped into *ins* if necessary. The simplex creates its own copy of the debug name given in the buffer *namein*.

```
Simplex(Basis& f)
```

Used to downcast a general basis to a simplex. Only succeeds if the general basis is holding either a simplex or a frame; the latter can be automatically cast to a simplex.

Miscellaneous

```
Simplex(void)
Simplex(Simplex& b)
Simplex& operator=(Simplex& b)
void debug_out(ostream& c, int indent)
```

Standard housekeeping functions.

4.3 Member Functions for Frame

Constructors

```
Frame(Basis& f)
```

Used to downcast a general basis to a frame. Only succeeds if the general basis is holding a frame.

```
Frame(char* namein, ASpace& ins, GeObList& t)
```

Builds a frame for the affine space *ins*, using the objects in the list t, after they are mapped into *ins* (or its tangent space) if necessary. The frame creates its own copy of the debug name given in the buffer *namein*. The last object in the list t is the point member of the frame.

```
Frame(Simplex& b, int n)
```

Builds a frame from the simplex b. The n^{th} point in the simplex becomes the point member of the frame.

Miscellaneous

```
Frame(void)
Frame(Frame& f)
Frame& operator=(Frame& f)
void debug_out(ostream& c, int indent)
```

Standard housekeeping functions.

4.4 Member Functions for VBasis

Constructors

```
VBasis(char* namein, GeOb& t)
```

If t is a vector tuple from a cartesian product vector space, this builds a vector basis from the tuple members. The tuple can be a mixture of vectors from a linearization space and affine vectors from a tangent space; the tuple elements are mapped into the appropriate space as needed. The basis creates its own copy of the debug name given in the buffer *namein*.

```
VBasis(char* name, VSpace& ins, GeObList& t)
```

Builds a vector basis for the vector space *ins*, using the objects in the list t, after they are mapped into *ins* if necessary. The basis creates its own copy of the debug name given in the buffer *namein*.

```
VBasis(Basis& f)
```

Used to downcast a general basis to a vector basis. Only succeeds if the general basis is holding a vector basis.

Miscellaneous

```
VBasis(void)
VBasis(VBasis& v)
VBasis& operator=(VBasis& b)
void debug_out(ostream& c, int indent)
```

Standard housekeeping functions.

4.5 Member Functions for HFrame

Constructors

`HFrame(char* namein, PSpace& ins, GeObList& t)`

Builds a projective frame for the projective space *ins*, using the objects in the list *t*, after they are mapped into *ins* if necessary. The projective frame creates its own copy of the debug name given in the buffer *namein*.

`HFrame(Basis& f)`

Used to downcast a general basis to a projective frame. Only succeeds if the general basis is holding a projective frame.

Miscellaneous

```
HFrame(void)
HFrame(HFrame& h)
HFrame& operator=(HFrame& h)
void debug_out(ostream& c, int indent)
```

Standard housekeeping functions.

5 GeOb

5.1 Member Functions for the Base Class GeOb

Typecasting (Including Single-Argument Constructors)

`GeOb(MultiMap& mp)`

If the multimap *mp* has zero arguments, it is cast into a geometric object in its range space. If it is instead a one-argument linear map into the space "Reals", it is cast into a vector (or affine vector) in the dual space of the domain.

`GeOb(Map& mp)`

If the domain subset of the map *mp* is a full subset of a vector space, and the range is the space "Reals", the map is cast into a vector (or affine vector) in the dual space of the domain.

`GeOb(Scalar v)`

The scalar *v* is automatically cast into a vector in the predefined space "Reals", using the standard basis "1".

`Scalar ToScalar(void)`

If this GeOb is a vector in the space "Reals", it is cast into a scalar, using the standard basis "1". This casting must be done explicitly (this avoids compilation ambiguity difficulties).

Interrogation Functions

`Space SpaceOf(void)`

Returns the space containing the geometric object.

`GeObType Holds(void)`

Returns the type of geometric object currently held by the GeOb.

`Boolean CanMapTo(GeObType t)`

Returns TRUE if and only if the geometric object can be successfully mapped into an object of type t, using the standard mappings.

`Boolean IsZeroVector(void)`

Returns TRUE if and only if the object is a zero vector.

`GeOb operator[](int n)`

Returns the n^{th} element of a vector or point tuple in a cartesian product space, using zero-based indexing. If this GeOb is not a vector, affine vector, or affine point, it is first mapped to a vector.

`GeObList TupleElements(void)`

Returns a list of all the elements of a vector or point tuple in a cartesian product space. If this GeOb is not a vector, affine vector, or affine point, it is first mapped to a vector.

Operations

Algebraic operations are implemented as friends due to the binary nature of the operators. Note that these operators will work with vector (and affine vector) equivalence class arguments. This is included for completeness; the randomization that occurs in mappings from vector (and affine vector) equivalence classes makes it unlikely that this feature will be useful.

`friend GeOb operator-(GeOb& th)`

If the object is a vector or affine vector, this negates it. Otherwise, it is mapped into a vector in the linearization space (or into a tangent space vector, if it is an affine vector equivalence class) prior to negation.

```
friend GeOb operator+(GeOb& th, GeOb& g)
friend GeOb operator-(GeOb& th, GeOb& g)
```

If the objects are vectors or affine vectors in the same space, these operators add or subtract them. Otherwise, the objects are first mapped into vectors in the linearization space (or into tangent space vectors, if the operation involves just affine vector equivalence classes and/or affine vectors).

```
friend GeOb operator*(GeOb& th, GeOb& g)
```

This operation of scalar multiplication is only legal if one of the geometric objects is a vector in the space "Reals". If the other object is not a vector or affine vector, it is first mapped into a vector in the linearization space (or into a tangent space vector, if it is an affine vector equivalence class) before the scalar multiplication is carried out.

```
friend GeOb operator/(GeOb& th, GeOb& g)
```

Just like the scalar multiplication operation described above, except that the second operand g is required to be a nonzero vector in the space "Reals".

```
Scalar Apply(GeOb& a)
```

Treat this GeOb as a linear functional and apply it to the object a in its dual space. One of the objects must be in either the linearization dual space or tangent dual space (if that object is a vector equivalence class, it is mapped to a vector in the space). The other object is first mapped, if necessary, to a vector or affine vector in the corresponding primal space.

```
GeOb Dual(void)
```

If this GeOb is a vector or affine vector, return the dual vector. Otherwise, it is mapped into a vector in the linearization space (or into a tangent space vector, if it is an affine vector equivalence class) prior to taking the dual. Only valid if the space is Euclidean.

```
GeOb MapTo(GeObType t)
```

Apply the standard mappings to this GeOb to convert it to a GeOb of type t, if possible.

```
GeOb SetTupleElement(int n, GeOb& g)
```

If this GeOb is not a vector, affine vector, or affine point, it is first mapped to a vector. This function returns a tuple that has the n^{th} element of this tuple changed to g, using zero-based indexing. The object g is first mapped into the corresponding space component of the cartesian product space, if necessary.

```
friend PPoint CrossRatio(AugScalar v, GeObList& g)
```

The list g must consist of three objects that can be cast into collinear points in the projective completion space. Given the list of points (A, B, C), and the cross ratio $v = (A, B; C, D)$, this routine returns the projective point D.

```
friend AugScalar CrossRatioCalc(GeObList& g)
```

The list g must consist of four objects that can be cast into collinear points in the projective completion space, where at least three are distinct. Given the list of points (A, B, C, D), this function returns the cross ratio $(A, B; C, D)$.

Miscellaneous

```
GeOb(void)
GeOb(GeOb& g)
GeOb& operator=(GeOb& g)
void debug_out(ostream& c, int indent)
```

Standard housekeeping functions.

5.2 Member Functions for Vector

Constructors

```
Vector(VBasis& b, ScalarList& a)
```

This creates a vector in the space containing the basis b with the coordinates given in the list of scalars.

```
Vector(VSpace& ins, GeObList& s)
Vector(VSpace& ins, GeOb& v1, GeOb& v2)
Vector(VSpace& ins, GeOb& v1, GeOb& v2, GeOb& v3)
```

These constructors build a vector tuple in the specified cartesian product vector space *ins*. The elements of the tuple are either given by the objects in the list s (for the first constructor), or individually (the other two constructors). The elements are first cast into vectors or affine vectors, if necessary.

Typecasting (Including Single-Argument Constructors)

```
Vector(GeOb& a)
```

Used to downcast an arbitrary geometric object to a vector. Standard maps will be used to map the GeOb to a vector; this will succeed if the mapping succeeds.

```
Vector(MultiMap& m)
Vector(Map& m)
Vector(Scalar s)
```

See the descriptions given in Section **??** for the GeOb single-argument constructors.

Mapping

```
Boolean CanMapTo(GeObType t)
GeOb MapTo(GeObType t)
```

See the description given in Section **??** for the base class (these are implemented as virtual functions).

Miscellaneous

```
Vector(void)
Vector(Vector& v)
Vector& operator=(Vector& s)
void debug_out(ostream& c, int indent)
```

Standard housekeeping functions.

5.3 Member Functions for AVector

Constructors

```
AVector(Basis& b, ScalarList& a)
```

This creates an affine vector specified by the coordinates a with respect to the basis b. The basis can either be a vector basis for the tangent space, or a simplex or frame for the associated affine space.

```
AVector(VSpace& ins, GeObList& s)
AVector(VSpace& ins, GeOb& v1, GeOb& v2)
AVector(VSpace& ins, GeOb& v1, GeOb& v2, GeOb& v3)
```

These constructors build an affine vector tuple in the specified cartesian product vector space *ins*. The elements of the tuple are either given by the objects in the list *s* (for the first constructor), or individually (the other two constructors). The elements are first cast into affine vectors, if necessary.

Typecasting (Including Single-Argument Constructors)

```
AVector(GeOb& a)
```

Used to convert an arbitrary geometric object to an affine vector. Standard maps will be used to map the GeOb to an affine vector; this will succeed if the mapping succeeds.

```
AVector(MultiMap& m)
AVector(Map& m)
```

See the descriptions given in Section **??** for the GeOb single-argument constructors.

Mapping

```
Boolean CanMapTo(GeObType t)
GeOb MapTo(GeObType t)
```

See the description given in Section **??** for the base class (these are implemented as virtual functions).

Miscellaneous

```
AVector(void)
AVector(AVector& v)
AVector& operator=(AVector& g)
void debug_out(ostream& c, int indent)
```

Standard housekeeping functions.

5.4 Member Functions for VectorEC

Typecasting (Including Single-Argument Constructors)

```
VectorEC(GeOb& a)
```

Used to convert an arbitrary geometric object to a vector equivalence class. Standard maps will be used to map the GeOb to a vector equivalence class; this will succeed if the mapping succeeds.

```
VectorEC(MultiMap& m)
VectorEC(Map& m)
```

See the descriptions given in Section **??** for the GeOb single-argument constructors. If m can be cast into a vector, these constructors will then attempt to convert it to a vector equivalence class.

Mapping

```
Boolean CanMapTo(GeObType t)
GeOb MapTo(GeObType t)
```

See the description given in Section **??** for the base class (these are implemented as virtual functions).

Miscellaneous

```
VectorEC(void)
VectorEC(VectorEC& v)
VectorEC& operator=(VectorEC& g)
void debug_out(ostream& c, int indent)
```

Standard housekeeping functions.

5.5 Member Functions for AVectorEC

Typecasting (Including Single-Argument Constructors)

```
AVectorEC(GeOb& a)
```

Used to convert an arbitrary geometric object to an affine vector equivalence class. Standard maps will be used to map the GeOb to a affine vector equivalence class; this will succeed if the mapping succeeds.

```
AVectorEC(MultiMap& m)
AVectorEC(Map& m)
```

See the descriptions given in Section **??** for the GeOb single-argument constructors. If m can be cast into an affine vector, these constructors will then attempt to map it to an affine vector equivalence class.

Mapping

```
Boolean CanMapTo(GeObType t)
GeOb MapTo(GeObType t)
```

See the description given in Section **??** for the base class (these are implemented as virtual functions).

Miscellaneous

```
AVectorEC(void)
AVectorEC(AVectorEC& v)
AVectorEC& operator=(AVectorEC& g)
void debug_out(ostream& c, int indent)
```

Standard housekeeping functions.

5.6 Member Functions for APoint

Constructors

```
APoint(Basis& s, ScalarList& a)
```

This creates an affine point specified by the coordinates a with respect to the basis s. The basis can either be a simplex or a frame.

```
APoint(ASpace& ins, GeObList& s)
APoint(ASpace& ins, GeOb& p1, GeOb& p2)
APoint(ASpace& ins, GeOb& p1, GeOb& p2, GeOb& p3)
```

These constructors build a point tuple in the specified cartesian product affine space *ins*. The elements of the tuple are either given by the objects in the list s (for the first constructor), or individually (the other three constructors). The elements are first cast into affine points, if necessary.

Typecasting (Including Single-Argument Constructors)

```
APoint(GeOb& a)
```

Used to convert an arbitrary geometric object to an affine point. Standard maps will be used to map the GeOb to an affine point; this will succeed if the mapping succeeds.

```
APoint(MultiMap& m)
APoint(Map& m)
```

See the descriptions given in Section **??** for the GeOb single-argument constructors. If m can be cast into some sort of geometric object, these constructors will then attempt to map it to an affine point using standard maps.

Mapping

```
Boolean CanMapTo(GeObType t)
GeOb MapTo(GeObType t)
```

See the description given in Section **??** for the base class (these are implemented as virtual functions).

Miscellaneous

```
APoint(void)
APoint(APoint& p)
APoint& operator=(APoint& g)
void debug_out(ostream& c, int indent)
```

Standard housekeeping functions.

5.7 Member Functions for PPoint

Constructors

```
PPoint(HFrame& s, ScalarList& a)
```

This creates a projective point specified by the homogeneous coordinates a with respect to the projective frame s.

Typecasting (Including Single-Argument Constructors)

```
PPoint(GeOb& a)
```

Used to convert an arbitrary geometric object to a projective point. Standard maps will be used to map the GeOb to a projective point; this will succeed if the mapping succeeds.

```
PPoint(MultiMap& m)
PPoint(Map& m)
```

See the descriptions given in Section **??** for the GeOb single-argument constructors. If m can be cast into some sort of geometric object, these constructors will then attempt to map it to a projective point using standard maps.

Mapping

`Boolean CanMapTo(GeObType t)`
`GeOb MapTo(GeObType t)`

See the description given in Section **??** for the base class (these are implemented as virtual functions).

Miscellaneous

```
PPoint(void)
PPoint(PPoint& v)
PPoint& operator=(PPoint& g)
void debug_out(ostream& c, int indent)
```

Standard housekeeping functions.

6 Map

6.1 Member Functions for the Base Class Map

Typecasting (Including Single-Argument Constructors)

`Map(MultiMap& m)`

If the multimap m has a simple vector or affine space as a domain (i.e. the domain cartesian product space has only one component space), this routine will convert it into a simple map.

`Map(GeOb& g)`

If the argument g is a vector or affine vector (or can be converted to a vector), the routine converts it to a map from the dual space of the vector to the reals.

Interrogation Functions

`SubSet Range(void)`

Returns the subset that is the range of the map.

`SubSet Domain(void)`

Returns the subset that is the domain of the map.

`MapType Holds(void)`

Returns the type of map currently held by the map.

`Boolean Invertible(void)`

Returns TRUE if and only if the map is invertible.

Functions to Apply Maps

`GeOb operator()(GeOb& v)`

Apply this map to the geometric object v and return the image in the range. If the argument is not in the domain space, an attempt is made to map it using the standard mappings.

Functions that Compose and Invert Maps

`Map Inv(void)`

Inverts this map. Only legal if the map is invertible.

`Map Compose(Map& m)`

Composes this map with the map m and returns the result. The range of map m, which is applied first, must be a subset of the domain of this map. Only valid if the two maps are the same type (e.g. both affine).

`ProjectiveMap ComposeProj(Map& m)`

Compose affine and projective maps to form a projective map. Any affine maps are first converted to projective maps using the following $InducedProjective()$ function.

Functions to Obtain Induced Maps

`ProjectiveMap InducedProjective(void)`

Only valid if this map is an invertible affine map between full affine subsets of affine spaces. This routine returns the corresponding projective map between the neighboring projective completion spaces.

`LinearMap InducedLinear(void)`

Only valid if this map is an affine map between full affine subsets of affine spaces. This routine returns the corresponding linear map between the neighboring linearization spaces.

`LinearMap Trans(void)`

Returns the transpose of this map. Only valid if this map is a linear map between full linear subsets of vector spaces.

`LinearMap AssocLinear(void)`

If this is an affine map from an affine subset of an affine space to an affine or linear subset of a linear or affine space, this routine returns the map induced on the tangent space of the domain space.

`Vector AssocDualVector(void)`

If this map is an affine functional (i.e. an affine map into the reals), this function returns the vector in the dual of the tangent space corresponding to the associated linear map.

Miscellaneous

```
Map(void)
Map(Map& m)
Map& operator=(Map& m)
void debug_out(ostream& c, int indent)
```

Standard housekeeping functions.

6.2 Member Functions for LinearMap

Constructors

`LinearMap(VBasis& b1, VBasis& b2)`

Builds an invertible linear map between two vector spaces that carries the vectors in basis $b1$ to the vectors in $b2$.

`LinearMap(VBasis& b, SubSet& s, GeObList& v)`

More general linear map creation routine. The domain of the map is a whole vector space, but the range can be a linear subset of a vector space. The objects in the list v are the images of the vectors in basis b; they are first mapped into the space of s, if necessary. The image objects do not have to be independent or span the subset s.

`LinearMap(SubSet& s, GeObList& v, VBasis& b)`

Similar to the previous linear map creation routine, except that the range of the map is a whole vector space, while the domain can be a linear subset of a vector space. The objects in the list v are the preimages of the vectors in basis b; they are first mapped into the space of s, if necessary. In this case, the image objects in v must be independent and span the subset s.

`LinearMap(SubSet& s1, GeObList& v1, SubSet& s2, GeObList& v2)`

Most general linear map creation routine, where both the range and domains are linear subsets of a vector space. The objects in the list $v1$ map to the objects in list $v2$; standard maps are first applied to the objects in the lists if necessary. The image objects in $v1$ must be independent and span the subset $s1$.

Typecasting (Including Single-Argument Constructors)

```
LinearMap(MultiMap& m)
LinearMap(GeOb& g)
```

See the descriptions given in Section **??** for the Map single-argument constructors.

```
LinearMap(Map& m)
```

Used to downcast a general map to a linear map. Only succeeds if the general map is holding a linear map.

Miscellaneous

```
LinearMap(void)
LinearMap(LinearMap& m)
LinearMap& operator=(LinearMap& s)
void debug_out(ostream& c, int indent)
```

Standard housekeeping functions.

6.3 Member Functions for AffineMap

Constructors

```
AffineMap(Basis& b1, Basis& b2)
```

Builds an invertible affine map between two affine spaces that carries the objects in basis $b1$ to the objects in $b2$. The bases can be either both simplices or both frames.

```
AffineMap(Simplex& b, SubSet& s, GeObList& v)
```

More general linear map creation routine. The domain of the map is a whole affine space, but the range can be either a linear subset of a vector space or an affine subset. Note that the basis for the domain space is restricted to be a simplex. The objects in the list v are the images of the objects in basis b; they are first mapped into the space of s, if necessary. The image objects do not have to be independent or span the subset s.

```
AffineMap(SubSet& s, GeObList& v, Simplex& b)
```

Similar to the previous affine map creation routine, except that the range of the map is a whole affine space, while the domain can be an affine subset. The objects in the list v are the preimages of the points in simplex b; they are first mapped into the space of s, if necessary. In this case, the image objects in v must be independent and span the subset s.

`AffineMap(SubSet& s1, GeObList& v1, SubSet& s2, GeObList& v2)`

Most general affine map creation routine, where the domain is some affine subset and the range is either a linear or affine subset. The objects in the list $v1$ map to the objects in list $v2$; standard maps are first applied to the objects in the lists if necessary. The image objects in $v1$ must be independent and span the subset $s1$.

Typecasting (Including Single-Argument Constructors)

`AffineMap(MultiMap& m)`

If the multimap m has an atomic affine space as a domain (i.e. the domain cartesian product space has only one component space), this routine will convert it into a simple affine map.

`AffineMap(Map& m)`

Used to downcast a general map to an affine map. Only succeeds if the general map is holding an affine map.

Miscellaneous

```
AffineMap(void)
AffineMap(AffineMap& m)
AffineMap& operator=(AffineMap& s)
void debug_out(ostream& c, int indent)
```

Standard housekeeping functions.

6.4 Member Functions for ProjectiveMap

Constructors

`ProjectiveMap(HFrame& b1, HFrame& b2)`

Builds an invertible projective map between two projective spaces that carries the projective points in the projective frame $b1$ to the projective points in $b2$.

`ProjectiveMap(HFrame& b, SubSet& s, GeObList& v)`

More general projective map creation routine, used to build invertible maps from a whole projective space to a projective subset. Since the only allowable method for creating noninvertible maps is to have a domain subspace with removed points, this routine requires the image objects v to be independent and span s. Note that s could be a projective subset of a vector space. The range subset s must be projective and cannot have removed points. The objects in the list v are the images of the objects in the frame b; they are first mapped into the space of s, if necessary.

`ProjectiveMap(SubSet& s, GeObList& v, HFrame& b)`

This constructor is used to build a projective map from a projective subset to a projective space. If the domain subset has removed points, this map will not be invertible. The objects in the list v are the preimages of the objects in the frame b; they are first mapped into the space of s, if necessary. The objects in v must be in general position and span the subset s.

`ProjectiveMap(SubSet& s1, GeObList& v1, SubSet& s2, GeObList& v2)`

Most general projective map creation routine, where the domain and range are projective subsets. If the domain subset $s1$ has removed points, this map will not be invertible. The range subset $s2$ cannot have removed points. The objects in the list $v1$ map to the objects in list $v2$; standard maps are first applied to the objects in the lists if necessary. The objects in both $v1$ and $v2$ must be in general position and span their respective subsets $s1$ and $s2$.

Typecasting (Including Single-Argument Constructors)

`ProjectiveMap(Map& m)`

Used to downcast a general map to a projective map. Only succeeds if the general map is holding a projective map.

Miscellaneous

```
ProjectiveMap(void)
ProjectiveMap(ProjectiveMap& m)
ProjectiveMap& operator=(ProjectiveMap& s)
void debug_out(ostream& c, int indent)
```

Standard housekeeping functions.

7 MultiMap

7.1 Member Functions for the Base Class MultiMap

Typecasting (Including Single-Argument Constructors)

`Scalar ToScalar(void)`

If this multimap has zero arguments, and the range space is the space "Reals", this routine converts it to a scalar, using the standard basis of "1". Note that this conversion must be done explicitly.

`MultiMap(Map& m)`

If the map m is linear, or is an affine map from an affine space, and if the domain is a full space, this routine casts it into a multimap.

`MultiMap(GeOb& g)`

If the argument g is a vector or affine vector (or can be converted to a vector), this routine casts it to a map from the dual space of the vector to the space "Reals", which is then in turn cast to a multimap. Note that there is no support for casting a GeOb into a zero-argument multimap.

Interrogation Functions

`Space RangeSpace(void)`

Returns the range space of this multimap.

`Space DomainSpace(void)`

Returns the domain space of this multimap.

`Boolean FullyEvaluated(void)`

Returns TRUE if and only if this multimap has been fully evaluated (i.e. it is a zero-argument map).

`MultiType Holds(void)`

Returns the type of multimap currently held by this multimap.

Functions to Apply Maps and do Partial Evaluation

`GeOb operator()(GeOb& t)`

Apply the multimap to the argument t, which is a tuple from the cartesian product domain space, and return the image in the range space.

```
MultiMap operator()(GeObList& t)
```

Apply the multimap to the list of arguments t where the n^{th} element of the list is a member of the n^{th} component space of the cartesian product domain space; standard maps are first applied to the arguments as needed. A list element that is a NULL_GEOB causes the routine to skip evaluation of the corresponding argument, permitting partial evaluation of the map. The routine returns the new multimap that results from this evaluation. This routine is not typically recommended for doing partial evaluation, as the system must create a new space to serve as the domain of the partially evaluated map (see the next function).

```
MultiMap operator()(Space& newdom, GeObList& t)
```

Like the above routine, but saves the system from having to create a new space. The space *newdom* is used as the domain for the partially evaluated multimap that is returned.

Miscellaneous

```
MultiMap(void)
MultiMap(MultiMap& v)
MultiMap& operator=(MultiMap& s)
void debug_out(ostream& c, int indent)
```

Standard housekeeping functions.

7.2 Member Functions for MLM

Constructors

```
MLM(VSpace& s1, IntList& symmetry, BasisList& bases,
    VSpace& s2, GeObList& vectors)
```

Creates a multilinear map from the cartesian product domain space $s1$ to the range space $s2$. The list of integers *symmetry* specifies the symmetry characteristics of the map. The basis list *bases* specifies one basis for each symmetry group in the domain, while the list *vectors* give the images in the range. See the thesis writeup for more detailed information and examples.

Typecasting (Including Single-Argument Constructors)

```
MLM(MultiMap& s)
```

Used to downcast a general multimap to a multilinear map. Only succeeds if the general multimap is holding a multilinear map.

```
MLM(Map& m)
MLM(GeOb& g)
```

See the descriptions given in Section **??** for the MultiMap single-argument constructors.

Miscellaneous

```
MLM(void)
MLM(MLM& m)
MLM& operator=(MLM& s)
void debug_out(ostream& c, int indent)
```

Standard housekeeping functions.

7.3 Member Functions for MAM

Constructors

```
MAM(ASpace& s1, IntList& symmetry, BasisList& simplices,
    Space& s2, GeObList& images)
```

Creates a multiaffine map from the cartesian product domain space $s1$ to the range space $s2$. The list of integers *symmetry* specifies the symmetry characteristics of the map. The simplex list *simplices* specifies one simplex for each symmetry group in the domain, while the list *images* give the images in the range. See the thesis writeup for more detailed information and examples.

Typecasting (Including Single-Argument Constructors)

```
MAM(MultiMap& s)
```

Used to downcast a general multimap to a multiaffine map. Only succeeds if the general multimap is holding a multiaffine map.

```
MAM(Map& m)
```

See the description given in Section **??** for the MultiMap single-argument constructors.

Miscellaneous

```
MAM(void)
MAM(MAM& m)
MAM& operator=(MAM& s)
void debug_out(ostream& c, int indent)
```

Standard housekeeping functions.

8 AugScalar

8.1 Member Functions for the Base Class AugScalar

Constructors

`AugScalar(Infnum n)`

Creates an augmented scalar with the value infinity when the argument n is equal to "INFINITY". CAUTION: INFINITY is implemented as an enumerated data type with the arbitrary integer value -23457.

Typecasting (Including Single-Argument Constructors)

`AugScalar(Scalar v)`

Creates an augmented scalar of value v.

`operator Scalar()`

Used for casting augmented scalars to scalars automatically.

Interrogation Functions

`Scalar Value(void)`

Return the value of the augmented scalar. Only valid if it is not equal to infinity.

`Boolean IsInfinity(void)`

Returns TRUE if and only if the augmented scalar has the value INFINITY.

Miscellaneous

```
AugScalar(void)
AugScalar(AugScalar& a)
AugScalar& operator=(AugScalar& a)
void debug_out(ostream& c, int indent)
```

Standard housekeeping functions.

9 Miscellaneous

9.1 Object and ErrorHandler Classes

To facilitate error reporting, every class in this package (except the error handler) is a subclass of the Object class. Every object knows how to print itself out, using the virtual function

```
void debug_out(ostream& c, int indent)
```

which prints out the contents of the object on output stream *c*. The printout is indented *indent* characters. The stream output operator "<<" is also defined for objects.

The error handler (there is a single predefined instance of the error handler class, "errh", provided with the system) is used to print out error messages and terminate the program. Member functions for the handler are of the form

```
void ErrorExit(char* errloc, char* descript, Object& o1)
```

where *errloc* contains the function signature for where the error occurred, *descript* gives a description for the error, and *o*1 is an object that will have its contents printed out. Member functions accepting from zero to five objects are provided. A special class ErrVal is also provided, which allows scalar and integer values to be passed to the error handler as Objects. ErrVal constructors are

```
ErrVal(char* message, Scalar val)
ErrVal(char* message, int val)
```

which permit values to be tagged with an explanatory message. There is also a special class ErrType, which allows a value of an enumerated types described in Section **??** to be passed to the error handler as well. To construct an ErrType, the user specifies a message, the value of the enumerated type, and a key to what the enumerated type is:

```
ErrVal(char* message, int val, EnumSet s)
```

9.2 Matrix and RowMatrix Classes

The geometry package is built on top of a matrix package that implements the Matrix and RowMatrix classes; a matrix is built up of row matrices. Since it is not intended for the user to be using this underlying support layer, a detailed description of the package will not be provided.

9.3 Booleans and Scalars

Booleans are implemented in this package as an enumerated data type that takes on the values TRUE and FALSE. Scalars are implemented as doubles.

9.4 Enumerated Types

Several enumerated data types are defined in the system. These types are:

SRel: LINEARIZATION, AFFINE, TANGENT, PROJECT_COMP, LIN_DUAL, TANG_DUAL, NO_RELATION, SAME_SPACE

SpaceType: NULL_SPACE, VEC_SPACE, AFF_SPACE, PROJ_SPACE, ANY_SPACE

BasisType: NULL_BASIS, SIMPLEX, FRAME, VBASIS, HFRAME, ANY_BASIS

GeObType: NULL_GEOB, VECTOR, AFF_POINT, AFF_VECTOR, VEC_EC, AFF_VEC_EC, PROJ_POINT, ANY_GEOB

MapType: NULL_MAP, LIN_MAP, AFF_MAP, PROJ_MAP, ANY_MAP

MultiType: NULL_MULTI, MULTI_LINEAR, MULTI_AFFINE, ANY_MULTI

SubSetType: NULL_SUBSET, LINEAR_SUBSET, AFFINE_SUBSET, PROJECTIVE_SUBSET, ANY_SUBSET

EnumSet: SPACETYPES, BASISTYPES, GEOBTYPES, MAPTYPES, MULTITYPES, SUBSETTYPES, SRELTYPES

Probably the most important type for the user to be familiar with is the SRel type, which is used to navigate around the six-space set.

Output functions have been implemented for each data type (except EnumSet) to facilitate easy output for debugging purposes. For example:

```
ostream& GeObTypeOut(ostream& c, GeObType t)
```