Chapter 3.
Classes

Topic: Ignore

The class interface is a set of C++ classes, including their public and protected data and methods (member functions), that an application can use directly to interact with ACIS. Developers may also derive their own classes from these classes to add application-specific functionality and data. Refer to the *3D ACIS Online Help User's Guide* for a description of the fields in the reference template.

ACIS_OBJECT

Class: Memory Management

Purpose: Provides a base class for class level memory management of ACIS classes.

Derivation: ACIS_OBJECT : –

SAT Identifier: None

Filename: base/baseutil/mmgr/mmgr.hxx

Description: The ACIS_OBJECT class provides a base class level interface to memory management by overriding the standard new and delete operators.

A simple new operator is provided for older compilers that do not support overloading of new and delete, and a decorated new operator that keeps track of the source file and line where the new was issued is provided for newer compilers.

Additional versions of new are provided for allocating arrays of instances as well as single instances.

Limitations: None

References: None

Data:

None

Constructor:

```
public: void* operator ACIS_OBJECT::new (
    size_t alloc_size,       // size of requested
                             // memory block
    AcisMemType alloc_type,  // eDefault, or
                             // eSession, or
                             // eDocument, or
                             // eTemporary
    const char* alloc_file,  // name of file in
                             // which new occurred
    int alloc_line,          // line of file in
                             // which new occurred
    int* alloc_file_index    // must always be
                             // &alloc_file_index
    );
```

New operator for single instances, with decorations to keep track of the file and line where the new was issued. This version is available only on newer compilers that permit overloading of new and delete, and when the MMGR_ENABLED compiler flag is given.

```
public: void* operator ACIS_OBJECT::new (
    size_t alloc_size,       // size of requested
                             // memory block
    size_t dummy             // argument forces
        = 0                  // pairing of scalar
                             // new with scalar
                             // delete
    );
```

New operator for single instances on older compilers where overloading of new and delete are not permitted, and the MMGR_FREELIST compiler flag is given.

```
public: void* operator new ACIS_OBJECT::[] (
    size_t alloc_size        // size of requested
                             // memory block
    );
```

New operator for arrays of instances on older compilers where overloading of new and delete are not permitted, and the MMGR_FREELIST compiler flag is given.

Destructor:

```
public: void operator ACIS_OBJECT::delete (
    void* alloc_ptr,        // pointer to memory
                            // block to delete
    size_t alloc_size       // size of memory block
                            // to delete
    );
```

Delete operator for single instances on older compilers where overloading of new and delete are not permitted, and the MMGR_FREELIST compiler flag is given.

```
public: void operator delete ACIS_OBJECT::[] (
    void* alloc_ptr         // pointer to memory
                            // block to delete
    );
```

Delete operator for arrays of instances.

Methods:

```
public: void* operator ACIS_OBJECT::new[] (
    size_t alloc_size,      // size of requested
                            // memory block
    AcisMemType alloc_type, // eDefault, or
                            // eSession, or
                            // eDocument, or
                            // eTemporary
    const char* alloc_file, // name of file in
                            // which new occurred
    int alloc_line,         // line of file in
                            // which new occurred
    int* alloc_file_index   // must always be
                            // &alloc_file_index
    );
```

New operator for arrays of instances, with decorations to keep track of the file and line where the new was issued. This version is available only on newer compilers that permit overloading of new and delete, and when the MMGR_ENABLED compiler flag is given.

Related Fncs:

acis_calloc, acis_free, acis_malloc, acis_realloc, acis_strdup

base_configuration

Class: Memory Management

Purpose: Initializes the Base library.

Derivation: base_configuration : –

SAT Identifier: None

Filename: base/baseutil/base.hxx

Description: Specifying a complex allocator or destructor disables all other configurable options. This class also contains data members that allow users to specify flags to turn on/off audit and freelist functionality. The defaults have been chosen to provide the expected desired behavior, so only users with special needs should need to explicitly change the settings.

The raw_allocator and raw_desctructor are function pointers within the base_configuration object. If provided when calling initialize_base, the ACIS Memory Manager will use these functions for all ACIS heap allocations (including memory required to implement freelisting and leak tracking).

This does not prevent ACIS from freelisting or managing memory. If these function pointers are provided and ACIS freelisting is enabled, the ACIS freelisting mechanism will use the raw_allocator and raw_destructor for managing memory required for freelisting. If user wishes to wholly replace ACIS memory management, there are two options:
1) provide a raw_allocator and raw_destructor, and set enable_freelists, enable_audit_logs, and enable_audit_leaks to FALSE; or
2) use the complex_allocator and complex_destructor.
If these functions are not provided, then by default, ACIS will use the C–runtime functions ”malloc” and ”free” as the raw_allocator and raw_destructor.

The complex_allocator and complex_destructor are function pointers that the customer may initialize within the base_configuration and pass to initialize_base. This allows the customer to intercept all heap allocations/deallocations without ACIS having any knowledge of the allocation/deallocation. In contrast to the raw_allocator, which describes the low–level allocation function which ACIS uses for allocating memory, the complex_allocator replaces the ACIS memory management system altogether. The prototype of the complex_allocator provides enough information to implement a memory management system as sophisticated as the ACIS memory manager. If the complex_allocator and complex_destructor are provided, all other base_configuration fields are ignored.

Limitations: None

References: None

Data:

```
public logical enable_audit_leaks;
```
Audit leaks flag.

```
public logical enable_audit_logs;
```
Audit logs flag.

```
public logical enable_freelists;
```
Freelists flag.

Constructor:

```
public: base_configuration::base_configuration ();
```

Default constructor.

Destructor:

None

Methods:

```
public: base_configuration::void (
    *complex_destructor)
    (void*,                 //pointer the caller
                            //wishes to deallocate
    AcisMemCall,            //style of deallocation
    size_t                  //number of bytes being
                            //deallocated (if known)
    );
```

Function pointer that the user may initialize within the base_configuration and pass to initialize_base. This allows the user to intercept all heap deallocations without ACIS having any knowledge of the deallocation.

```
public: base_configuration::void (
    *raw_destructor)(void*  //pointer the caller
                            //wishes to deallocate
    );
```

Function pointer within the base_configuration object for memory deallocation. Used for managing memory required for freelisting and leak tracking.

```
public: base_configuration::void* (
    *complex_allocator)(size_t,//number of bytes to
                               //allocate
    AcisMemType,            //classify the allocation
    AcisMemCall,            //style of allocation
    const char*,            //source of the
                            //allocation
    int,                    //line number of the
                            //allocation
    int*                    //pointer to storage for
                            //a file-specific index
                            //for each file that
                            //allocates memory
    );
```

Function pointer that the user may initialize within the base_configuration and pass to initialize_base. This allows the user to intercept all heap allocations without ACIS having any knowledge of the allocation.

```
public: base_configuration::void* ( //pointer
                                    //returned,
                                    //satisfying the
                                    //allocation request
                                    // or NULL if it is
                                    // unable to satisfy
                                    // the request
    *raw_allocator)
    (size_t                 //number of bytes to
                            //allocate
    );
```

Function pointer within the base_configuration object for memory allocation. Used for managing memory required for freelisting and leak tracking.

Related Fncs: initialize_base, terminate_base

SPAbox

Class: Mathematics

Purpose: Represents a bounding box.

Derivation: SPAbox : ACIS_OBJECT : –

SAT Identifier: None

Filename: base/baseutil/vector/box.hxx

Description: This class represents a bounding box. It is implemented as an axis-dependent, axis-aligned rectangular box, given by a triple of real intervals.

ACIS boxes major model entities for algorithm efficiency by constructing a simple bounding shape to surround the model entity (as closely as reasonably possible). When two entities are tested for interaction, the boxes can be tested first to filter out obviously disjoint cases.

The major items boxed are body, shell, and face. There is an additional model entity, the subshell, that exists solely to provide more efficient box testing. When sensible, a shell containing many faces is subdivided spatially into n subshells of faces. Each subshell fills approximately 1/n of the space filled by the original shell box. The subshells can in turn be subdivided if they contain a sufficient number of faces. A shell or subshell so subdivided can also contain faces directly, when these faces span the majority of the original box, and do not fit into any subshell box. The present algorithm for this subdivision limits *n* to 2.

Boxes are axis-aligned rectangular parallelepipeds. Subshell, shell, and body boxes are obtained by determining the overall limits of the boxes of the entities making up the body, shell, or subshell. When two bodies are compared, one must be transformed into the coordinate system of the other in order for the comparison to take place. For preliminary testing, each box is transformed and then boxed in the new coordinate system. This is not optimal, but it is relatively quick.

Boxes are computed only when needed, and changed entities merely require the existing box (if any) to be deleted; however, after a box is computed it is saved for later reuse. Boxes are not logged for roll back purposes, nor are they saved to a disk file. A box pointer in a roll back record is always set to NULL. After a roll back, such boxes must be recomputed.

Limitations: None

References: BASE SPAinterval

Data:

None

Constructor:

```
public: SPAbox::SPAbox ();
```

C++ allocation constructor requests memory for this object but does not populate it.

```
public: SPAbox::SPAbox (
    SPAbox const& old       // given box
    );
```

C++ copy constructor requests memory for this object and populates it with the data from the object supplied as an argument.

```
public: SPAbox::SPAbox (
    SPAinterval const&,     // extent in x-direction
    SPAinterval const&,     // extent in y-direction
    SPAinterval const&      // extent in z-direction
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

```
public: SPAbox::SPAbox (
    SPAposition const&      // given point
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

```
public: SPAbox::SPAbox (
    SPAposition const&,     // first point
    SPAposition const&      // second point
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

Destructor:

None

Methods:

```
public: logical SPAbox::bounded () const;
```

Returns TRUE if the coordinate ranges are finite or FALSE otherwise.

```
public: logical SPAbox::bounded_above () const;
```

Returns TRUE if the coordinate ranges are either finite or finite above or FALSE otherwise.

```
public: logical SPAbox::bounded_below () const;
```

Returns TRUE if the coordinate ranges are either finite or finite below or FALSE otherwise.

```
public: SPAposition SPAbox::corner (
    int index               // any value from 0 to 7
    ) const;
```

Returns the corners of the box – labeled from 0 to 7 where the corners correspond to writing the index in binary as x * 4 + y * 2 + z and letting zero corresponds to the low values, getting position (x, y, z).

```
public: void SPAbox::debug (
    char const*,            // title
    FILE*                   // file name
        = debug_file_ptr
    ) const;
```

Outputs a title and information about the box to the debug file or to the specified file.

```
public: logical SPAbox::empty () const;
```

Tests if the box is empty.

```
public: logical SPAbox::finite () const;
```

Returns TRUE if the coordinate ranges are finite or FALSE otherwise.

```
public: logical SPAbox::finite_above () const;
```

Returns TRUE if the coordinate ranges are finite above or FALSE otherwise.

```
public: logical SPAbox::finite_below () const;
```

Returns TRUE if the coordinate ranges are finite below or FALSE otherwise.

```
public: SPAposition SPAbox::high () const;
```

Extracts the high end of the leading diagonal of the box.

```
public: logical SPAbox::infinite () const;
```

Returns TRUE if any of the coordinate ranges is infinite or FALSE otherwise.

```
public: SPAposition SPAbox::low () const;
```

Extracts the low end of the leading diagonal of the box.

```
public: SPAposition SPAbox::mid () const;
```

Extracts the middle of the leading diagonal of the box.

```
public: SPAbox& SPAbox::operator&= (
    SPAbox const&           // given box
    );
```

Limits one box by another; i.e., this method forms the intersection of this box with the given box, which results in this box.

```
public: SPAbox& SPAbox::operator*= (
    double                  // scale factor
    );
```

Scales a box by the given factor. This results in a box in the new coordinate system that is sufficient to enclose the true transformed box.

```
public: SPAbox& SPAbox::operator*= (
    SPAmatrix const&        // given matrix
    );
```

Transforms a box by the given matrix. This results in a box in the new coordinate system that is sufficient to enclose the true transformed box.

```
public: SPAbox& SPAbox::operator*= (
    SPAtransf const&        // given transformation
    );
```

Transforms a box by the given transform. This results in a box in the new coordinate system that is sufficient to enclose the true transformed box.

```
public: SPAbox& SPAbox::operator+= (
    SPAvector const&        // vector
    );
```

Translates a box.

```
public: SPAbox& SPAbox::operator–= (
    SPAvector const&        // vector
    );
```

Translates a box.

```
public: logical SPAbox::operator<< (
    SPAbox const& b         // given box
    ) const;
```

Determines if the given box encloses this box.

```
public: SPAbox& SPAbox::operator/= (
    double                  // double
    );
```

Scales a box.

```
public: logical SPAbox::operator>> (
    SPAbox const&           // given box
    ) const;
```

Determines if this box entirely encloses the given box. This method returns TRUE if this box is NULL or if the given box is strictly within this box or within this box enlarged by SPAresabs in all six directions (+x, –x, +y, –y, +z, –z). Otherwise, or if the given box is NULL, this method returns FALSE.

```
public: logical SPAbox::operator>> (
    SPAposition const&      // given point
    ) const;
```

Determines if this box entirely encloses the given point. This method returns TRUE if this box is NULL or if the given point is strictly within this box or within this box enlarged by SPAresabs in all six directions (+x, –x, +y, –y, +z, –z). Otherwise, or if the given point is NULL, this method returns FALSE.

```
public: SPAbox& SPAbox::operator|= (
    SPAbox const&           // given box
    );
```

Compounds one box into another; i.e., this method extends this box until it also encloses the given box.

```
public: logical SPAbox::unbounded () const;
```

Returns TRUE if any of the coordinate ranges is infinite or FALSE otherwise.

```
public: logical SPAbox::unbounded_above () const;
```

Returns TRUE if either any of the coordinate ranges is infinite or the coordinate ranges are finite below or FALSE otherwise.

```
public: logical SPAbox::unbounded_below () const;
```

Returns TRUE if either any of the coordinate ranges is infinite or the coordinate ranges are finite above or FALSE otherwise.

```
public: SPAinterval SPAbox::x_range () const;
```

Retrieves the *x*-coordinate range.

```
public: SPAinterval SPAbox::y_range () const;
```

Retrieves the *y*-coordinate range.

```
public: SPAinterval SPAbox::z_range () const;
```

Retrieves the *z*-coordinate range.

Related Fncs:

enlarge_box

```
friend: SPAbox operator& (
    SPAbox const&,          // first box
    SPAbox const&           // second box
    );
```

Finds the overlap of two boxes; i.e., this method finds the intersection.

```
friend: SPAbox operator/ (
    SPAbox const&,          // given box
    double                  // scale factor
    );
```

Scales a box.

```
friend: SPAbox operator* (
    SPAbox const&,          // given box
    double                  // scale factor
    );
```

Scales a box.

```
friend: SPAbox operator* (
    SPAbox const&,          // given box
    SPAmatrix const&        // matrix
    );
```

Transforms a box by the given matrix. This results in a box in the new coordinate system that is sufficient to enclose the true transformed box.

```
friend: SPAbox operator* (
    SPAbox const&,          // given box
    SPAtransf const&        // transformation
    );
```

Transforms a box by the given transform. This results in a box in the new coordinate system that is sufficient to enclose the true transformed box.

```
friend: SPAbox operator* (
    double,                 // scale factor
    SPAbox const&           // given box
    );
```

Scales a box.

```
friend: SPAbox operator+ (
    SPAbox const&,          // given box
    SPAvector const&        // vector
    );
```

Translates a box.

```
friend: SPAbox operator+ (
    SPAvector const&,       // vector
    SPAbox const&           // given box
    );
```

Translates a box.

```
friend: SPAbox operator– (
    SPAbox const&,          // given box
    SPAvector const&        // vector
    );
```

Translates a box.

```
friend: SPAbox operator| (
    SPAbox const&,          // first box
    SPAbox const&           // second box
    );
```

Creates a box that encloses the two given boxes.

```
friend: SPAinterval operator% (
    SPAbox const&,          // given box
    SPAunit_vector const&   // direction
    );
```

Finds the extent of the box along the given direction.

```
friend: SPAinterval operator% (
    SPAunit_vector const&,   // direction
    SPAbox const&            // given box
    );
```

Finds the extent of the box along the given direction.

```
friend: logical operator&& (
    SPAbox const&,           // first box
    SPAbox const&            // second box
    );
```

Determines whether two boxes overlap. This method returns TRUE if either box is NULL or if all the intervals of one box overlap the corresponding intervals of the other box.

```
friend: logical operator<< (
    SPAposition const& p,    // position
    SPAbox const& b          // given box
    );
```

Determines if a given box encloses a given position.

complex_number

Class: Mathematics

Purpose: Creates a data structure for the manipulation of complex number.

Derivation: complex_number : ACIS_OBJECT : –

SAT Identifier: None

Filename: base/baseutil/vector/complex.hxx

Description: This is a C++ class that holds two doubles, which are meant to reflect a real and imaginary part. This class has contains methods for all of the traditional overload operations.

The complex number class has the overloaded C++ addition, subtraction, multiplication, and division operators.

```
complex_number a(1,2);
complex_number b(3,4);
complex_number c, d;
c = a + b;      // c = 4 + 6i
d = a * b;      // d = –5 + 10i
```

Limitations: None

References: None

Data:

```
public double im;
```
The imaginary component of the complex number.

```
public double re;
```
The real component of the complex number.

Constructor:

```
public: complex_number::complex_number ();
```

C++ allocation constructor requests memory for this object but does not populate it.

```
public: complex_number::complex_number (
    double re               // real component
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

```
public: complex_number::complex_number (
    double re,              // real component
    double im               // imaginary component
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

Destructor:

None

Methods:

```
public: double complex_number::angle () const;
```

Returns a number that represents the polar coordinate angle in radians. The angle is counter clockwise from the real axis to the ray formed by the origin and the complex number. See also radius method.

```
public: double complex_number::max_coord () const;
```

Returns the absolute value of the coordinate whose absolute value is maximum. This is its maximum metric distance from the origin.

```
public: complex_number complex_number::operator/ (
    complex_number c         // second number
    );
```

This performs complex number division.

```
public: complex_number complex_number::operator* (
    complex_number c         // second number
    );
```

This performs complex number multiplication.

```
public: complex_number complex_number::operator+ (
    complex_number c         // second number
    );
```

This performs complex number addition.

```
public: complex_number complex_number::operator– (
    complex_number c         // second number
    );
```

This performs complex number subtraction.

```
public: double complex_number::radius () const;
```

Returns the Euclidian distance of the complex number to the origin, also represented by the square root of the summation of the real portion squared and the imaginary portion squared. See also angle method.

```
public: complex_number* complex_number::root (
    int n                    // which root to find
    ) const;
```

Returns an array of complex numbers which represent its *n*th's roots.

```
public: void complex_number::set (
    double re,               // real component
    double im                // imaginary component
    );
```

Changes the real and imaginary components in this instance of a complex number.

Related Fncs:

None

enum_table

Class: SAT Save and Restore

Purpose: Defines objects for storing the mapping between the enum values and their string representation.

Derivation: enum_table : ACIS_OBJECT : –

SAT Identifier: None

Filename: base/baseutil/mmgr/enum_tbl.hxx

Description: An enumeration table stores a mapping between the enum values and their string representation. The mapping is used when writing enumerated values to a sat file in their string representation, or when reading strings from a SAT file that need to be converted back to enumerated values.

Limitations: None

References: None

Data:

None

Constructor:

```
public: enum_table::enum_table (
    const enum_entry *ent    // initialization value
    );
```

C++ constructor, creates an enum_table object and initializes it with the specified enum entry.

Destructor:

None

Methods:

```
public: char const* enum_table::string (
    int value                // input value
    ) const;
```

Return string corresponding to given value. If no match, returns NULL.

```
public: int enum_table::value (
    char const *string      // input string
    ) const;
```

Return value corresponding to given string. If no match, value in terminating entry (generally –9999) is returned.

Related Fncs:

None

error_info

Class: Error Handling

Purpose: Defines objects for returning ACIS error information.

Derivation: error_info : ACIS_OBJECT : –

SAT Identifier: None

Filename: base/baseutil/errorsys/err_info.hxx

Description: Objects derived from this base class are used to return information in the outcome class. These objects are specifically designed so that following an error, APIs can automatically return additional information to the user, simply by changing the sys_error call.

Although no restriction is placed on the information contained by an error_info object, new ENTITYs will be lost during roll back.

Error System Process

1. At the start of each API, a global variable pointer to an error_info object is set to NULL.
2. Before sys_error is called, the global pointer is set to contain the relevant error_info object.
3. At the end of the API, before the outcome is returned, the global variable is examined, and if nonempty, the error_info is added to the outcome.

Two overloaded versions of the function sys_error set a global pointer to an error_info object. One version is passed an error_info object, and the other creates a standard_error_info object when sys_error is passed one or two ENTITYs. The standard_error_info class is derived from error_info.

In the Local Ops, Remove Faces, and Shelling Components, the error_info object returns an ENTITY that further specifies where the local operation first fails, when such information is available. A standard_error_info object is adequate for use in these components, and more detailed information could be returned, if necessary, by deriving a new class.

Limitations: None

References: None

Data:

None

Constructor:

```
public: error_info::error_info ();
```

C++ constructor, creating an error_info.

Destructor:

```
public: virtual error_info::~error_info ();
```

C++ destructor, deleting an error_info.

Methods:

```
public: void error_info::add ();
```

Increments the use count.

```
public: static int error_info::id ();
```

Identifies the error object.

```
public: void error_info::remove ();
```

Decrements the use count.

```
public: virtual int error_info::type () const;
```

Returns the string “error_info”.

Related Fncs:

None

error_list_info

Class: Error Handling

Purpose: Chains a list of error_infos together.

Derivation: error_list_info : error_info : ACIS_OBJECT : –

SAT Identifier: None

Filename: base/baseutil/errorsys/err_list.hxx

Description: This class adds chaining capability to the basic error_info class. To make it fit properly into the use counting scheme, being referred to in a next_ptr counts as a use. Note that decrementing the use of the head of a list will potentially clean up the entire remainder of the list (if none have any other recorded uses). Any function that returns an error_list_info (with a view, perhaps, to the caller passing it to sys_error) should arrange the head of the list to have a use of 0. All the remaining items down the list will typically have a use of 1, being each referred to by just the one previous list item.

Limitations: None

References: None

Data:

None

Constructor:

```
public: error_list_info::error_list_info ();
```

Default constructor. Sets the next_ptr to NULL.

Destructor:

```
public: virtual error_list_info::~error_list_info ();
```

Default destructor. Removes a “use” from next_ptr.

Methods:

```
public: error_list_info* error_list_info::next ()
const;
```

Returns the next_ptr.

```
public: void error_list_info::set_next (
    error_list_info*         // list pointer
    );
```

Set next_ptr and juggle use counts.

Related Fncs:

None

exit_callback

Class: Callbacks

Purpose: Executes standard exit for ACIS.

Derivation: exit_callback : toolkit_callback : –

SAT Identifier: None

Filename: base/baseutil/acisio/acexit.hxx

Description: The exit_callback implements standard exit for ACIS. Any time a standard C exit call is made within ACIS, that call is redirected through this callback class. At that point the platform specific implementation of exit_callback obtains the exit from its natural exit stream.

Limitations: None

References: None

Data:

None

Constructor:

None

Destructor:

```
protected: virtual exit_callback::~exit_callback ();
```

C++ destructor, deleting an exit_callback.

Methods:

```
public: virtual logical exit_callback::execute (
    int                     // type of callback
    );
```

Returns FALSE if the standard exit should not be called after all exit callbacks are called.

Related Fncs:

None

input_callback

Class: Callbacks

Purpose: Creates the standard input for ACIS.

Derivation: input_callback : toolkit_callback : –

SAT Identifier: None

Filename: base/baseutil/acisio/acinput.hxx

Description: The input_callback implements standard input for ACIS. Any time a standard C input call is made within ACIS, that call is redirected through this callback class. At that point the platform specific implementation of input_callback obtains the input from its natural input stream.

Limitations: None

References: None

Data:

```
protected FILE *fp;
```

Pointer to the input file.

Constructor:

```
public: input_callback::input_callback (
    FILE* in_fp             // file pointer
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

Destructor:

```
protected: virtual
    input_callback::~input_callback ();
```

C++ destructor, deleting an input_callback.

Methods:

```
public: virtual void input_callback::do_clearerr ();
```

Clears the end of file and error indicators.

```
public: virtual int input_callback::do_feof ();
```

Returns a non–zero if end of file reached.

```
public: virtual int input_callback::do_ferror ();
```

Returns a non–zero if an error has occurred.

Clears end of file and error indicators.

```
public: virtual int input_callback::do_getc ();
```

Gets input character from input stream.

```
public: virtual int input_callback::do_ungetc (
    int c                   // character to push back
    );
```

Pushes character back onto input stream.

Related Fncs:

None

SPAinterval

Class: Mathematics

Purpose: Records an interval on a line.

Derivation: SPAinterval : ACIS_OBJECT : –

SAT Identifier: None

Filename: base/baseutil/vector/interval.hxx

Description: This class records an interval on the real line, i.e., a one dimensional region. It is implemented as an ordered pair of reals, together with a flag that indicates whether each end is bounded or not. This allows the representation and manipulation of finite, infinite, semi-infinite, and empty intervals. The boundary value at an unbounded end is irrelevant.

Limitations: None

References: by BASE SPAbox, SPApar_box

Data:

None

Constructor:

```
public: SPAinterval::SPAinterval ();
```

C++ allocation constructor requests memory for this object but does not populate it.

```
public: SPAinterval::SPAinterval (
    double d                // double value
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

Constructs a zero-length interval from one double value.

```
public: SPAinterval::SPAinterval (
    double,                 // double value
    double                  // double value
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

Constructs an interval from two values. The arguments do not need to be in ascending sequence. The constructor checks and adjusts the argument sequence.

```
public: SPAinterval::SPAinterval (
    interval_type,              // interval type
    double const&               // double value
        = * (double* ) NULL_REF,
    double const&               // double value
        = * (double* ) NULL_REF
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

Constructs a bounded or unbounded interval from a describer and up to two values. If the given interval type is interval_unknown, the type derives from the presence or absence of the real arguments (absence means infinite at the appropriate end); otherwise, the type is as given. Any required bound is taken from the arguments. If there are two real arguments, they are low/high. If there is one real argument, it is used for both ends.

Destructor:

None

Methods:

```
public: logical SPAinterval::bounded () const;
```

Determines if an interval is bounded above and below.

```
public: logical SPAinterval::bounded_above () const;
```

Determines if an interval is bounded above.

```
public: logical SPAinterval::bounded_below () const;
```

Determines if an interval is bounded below.

```
public: void SPAinterval::debug (
    FILE*                   // file name
        = debug_file_ptr
    ) const;
```

Outputs the details of an interval to the debug file or to the specified file.

```
public: void SPAinterval::debug_str (
    char*                   // string
    ) const;
```

Concatenates the value of interval to the passed string.

```
public: logical SPAinterval::empty () const;
```

Determines if an interval is empty.

```
public: double SPAinterval::end_pt () const;
```

Returns the end point of the interval. This method is only meaningful if the relevant ends are bounded; if the upper end is not bounded, there is no error.

```
public: logical SPAinterval::finite () const;
```

Determines if an interval is finite.

```
public: logical SPAinterval::finite_above () const;
```

Determines if an interval is finite above.

```
public: logical SPAinterval::finite_below () const;
```

Determines if an interval is finite below.

```
public: logical SPAinterval::infinite () const;
```

Determines if an interval is infinite.

```
public: double SPAinterval::interpolate (
    double                  // double value
    ) const;
```

Interpolate within the interval. This method returns:

```
(1 – parameter) * low–end + parameter * high–end
```

for the given parameter. This method is only meaningful if the relevant ends are bounded; if the relevant ends are not bounded, there is no error.

```
public: double SPAinterval::length () const;
```

Returns the difference between the high and low ends of the interval. By historical convention, both empty and infinite return negative values (since formerly an infinite parameter range returned empty), so this is retained. For convenient distinction, empty returns exactly –1.0, and infinite (or semi-infinite) returns exactly –2.0.

```
public: double SPAinterval::mid_pt () const;
```

Returns the middle point of the interval. This method is only meaningful if the relevant ends are bounded; if the relevant ends are not bounded, there is no error.

```
public: SPAinterval& SPAinterval::negate ();
```

Negates an interval in place.

```
public: SPAinterval& SPAinterval::operator&= (
    SPAinterval const&      // interval
    );
```

Finds the interval of overlap (the intersection of the two intervals).

```
public: SPAinterval& SPAinterval::operator*= (
    double                  // double value
    );
```

Multiplies an interval by a scalar (it multiplies both end values by a scalar value).

```
public: SPAinterval& SPAinterval::operator+= (
    double d                // double value
    );
```

Adds a scalar value to both ends of an interval, shifting the interval by the value.

```
public: SPAinterval& SPAinterval::operator+= (
    SPAinterval const&      // interval
    );
```

Adds two intervals together.

```
public: SPAinterval& SPAinterval::operator-= (
    double d                // double value
    );
```

Subtracts a scalar value from both ends of an interval, shifting the interval by the value.

```
public: SPAinterval& SPAinterval::operator-= (
    SPAinterval const&      // interval
    );
```

Subtracts an interface from an interval.

```
public: logical SPAinterval::operator<< (
    SPAinterval const& i    // interval
    ) const;
```

Inverses the given interval. Determine whether an interval is entirely enclosed within another interval. The given method returns TRUE if this interval is NULL, FALSE if this interval is NULL; otherwise, it returns TRUE if low end of this interval exceeds low end of this interval (less SPAresabs) and high end of this interval is less than high end of this interval (plus SPAresabs).

```
public: SPAinterval& SPAinterval::operator/= (
    double                  // double value
    );
```

Divides an interval (end value) by a scalar (value).

```
public: logical SPAinterval::operator>> (
    double                  // point
    ) const;
```

Determines whether a point lies within an interval. This method returns TRUE if this interval is NULL or if given value lies within this interval (expanded by SPAresabs at each end).

```
public: logical SPAinterval::operator>> (
    SPAinterval const&      // interval
    ) const;
```

Determine whether an interval is entirely enclosed within another interval. This method returns TRUE if this interval is NULL, FALSE if given interval is NULL; otherwise, it returns TRUE if low end of given interval exceeds low end of this interval (less SPAresabs) and high end of given interval is less than high end of this interval (plus SPAresabs).

```
public: SPAinterval& SPAinterval::operator|= (
    SPAinterval const&      // interval
    );
```

Constructs an interval containing two intervals (returns the union of both intervals).

```
public: logical SPAinterval::scalar () const;
```

Determine if the interval is a scalar (consisting of a single point).

```
public: double SPAinterval::start_pt () const;
```

Returns the start point of the interval. This method is only meaningful if the relevant ends are bounded; if the lower end is not bounded, there is no error.

```
public: interval_type SPAinterval::type () const;
```

Returns the type of interval_type.

```
public: logical SPAinterval::unbounded () const;
```

Determines if an interval is unbounded (not a finite interval).

```
public: logical SPAinterval::unbounded_above ()
const;
```

Determines if an interval is unbounded above.

```
public: logical SPAinterval::unbounded_below ()
const;
```

Determines if an interval is unbounded below.

Related Fncs:

None

```
friend: SPAinterval operator& (
    SPAinterval const&,      // interval
    SPAinterval const&       // interval
    );
```

Finds the interval of overlap.

```
friend: SPAinterval operator/ (
    SPAinterval const&,      // interval
    double                   // scalar value
    );
```

Divide an interval by a scalar.

```
friend: SPAinterval operator* (
    double,                  // scalar value
    SPAinterval const&       // interval
    );
```

Multiply a scalar by an interval.

```
friend: SPAinterval operator* (
    SPAinterval const&,      // interval
    double                   // scalar value
    );
```

Multiply an interval by a scalar.

```
friend: SPAinterval operator+ (
    double d,               // double
    SPAinterval const& i    // interval
    );
```

Add a double and an interval.

```
friend: SPAinterval operator+ (
    SPAinterval const&,     // interval
    SPAinterval const&      // interval
    );
```

Add two intervals together.

```
friend: SPAinterval operator+ (
    SPAinterval const& i,   // interval
    double d                // double
    );
```

Add an interval and a double.

```
friend: SPAinterval operator– (
    double d,               // double
    SPAinterval const& i    // interval
    );
```

Subtract an interval from a double.

```
friend: SPAinterval operator– (
    SPAinterval const&,     // interval
    SPAinterval const&      // interval
    );
```

Subtract two intervals.

```
friend: SPAinterval operator– (
    SPAinterval const& i,   // interval
    double d                // double
    );
```

Subtract a double from an interval.

```
friend: SPAinterval operator– (
    SPAinterval const&      // interval
    );
```

Negates an interval.

```
friend: SPAinterval operator| (
    SPAinterval const&,     // interval
    SPAinterval const&      // interval
    );
```

Construct an interval containing two intervals.

```
friend: logical operator!= (
    SPAinterval const& i1,  // interval
    SPAinterval const& i2   // interval
    );
```

Equality operator for determining whether two intervals are identical. The criteria are strict, so this operator should not be used when arithmetic equality is intended.

```
friend: logical operator&& (
    SPAinterval const&,     // interval
    SPAinterval const&      // interval
    );
```

Determine whether two intervals overlap. We test with respect to SPAresabs, to accommodate touching cases.

```
friend: logical operator< (
    double d,               // double
    SPAinterval const& i    // interval
    );
```

Arithmetic comparison. The meaning in each case is that every number in the first interval bears the given relationship (less than) with every number in the second, where a double is treated as an interval containing just that number.

```
friend: logical operator< (
    SPAinterval const& i,    // interval
    double d                 // double
    );
```

Arithmetic comparison. The meaning in each case is that every number in the first interval bears the given relationship (less than) with every number in the second, where a double is treated as an interval containing just that number.

```
friend: logical operator< (
    SPAinterval const& i1,   // interval
    SPAinterval const& i2    // double
    );
```

Arithmetic comparison. The meaning in each case is that every number in the first interval bears the given relationship (less than) with every number in the second, where a double is treated as an interval containing just that number.

```
friend: logical operator<< (
    double r,                // value
    SPAinterval const&  i    // interval
    );
```

Inverse of operator>> function.

```
friend: logical operator<= (
    double d,                // double
    SPAinterval const& i     // interval
    );
```

Arithmetic comparison. The meaning in each case is that every number in the first interval bears the given relationship (less than or equal to) with every number in the second, where a double is treated as an interval containing just that number.

```
friend: logical operator<= (
    SPAinterval const& i,    // interval
    double d                 // double
    );
```

Arithmetic comparison. The meaning in each case is that every number in the first interval bears the given relationship (less than or equal to) with every number in the second, where a double is treated as an interval containing just that number.

```
friend: logical operator<= (
    SPAinterval const& i1,  // interval
    SPAinterval const& i2   // interval
    );
```

Arithmetic comparison. The meaning in each case is that every number in the first interval bears the given relationship (less than or equal to) with every number in the second, where a double is treated as an interval containing just that number.

```
friend: logical operator== (
    SPAinterval const&,     // interval
    SPAinterval const&      // interval
    );
```

Equality operator for determining whether two intervals are identical. The criteria are strict, so this operator should not be used when arithmetic equality is intended.

```
friend: logical operator> (
    double d,               // double
    SPAinterval const& i    // interval
    );
```

Arithmetic comparison. The meaning in each case is that every number in the first interval bears the given relationship (greater than) with every number in the second, where a double is treated as an interval containing just that number.

```
friend: logical operator> (
    SPAinterval const& i,   // interval
    double d                // double
    );
```

Arithmetic comparison. The meaning in each case is that every number in the first interval bears the given relationship (greater than) with every number in the second, where a double is treated as an interval containing just that number.

```
friend: logical operator> (
    SPAinterval const& i1,  // interval
    SPAinterval const& i2   // interval
    );
```

Arithmetic comparison. The meaning in each case is that every number in the first interval bears the given relationship (greater than) with every number in the second, where a double is treated as an interval containing just that number.

```
friend: logical operator>= (
    double d,               // double
    SPAinterval const& i    // interval
    );
```

Arithmetic comparison. The meaning in each case is that every number in the first interval bears the given relationship (greater than or equal to) with every number in the second, where a double is treated as an interval containing just that number.

```
friend: logical operator>= (
    SPAinterval const& i,   // interval
    double d                // double
    );
```

Arithmetic comparison. The meaning in each case is that every number in the first interval bears the given relationship (greater than or equal to) with every number in the second, where a double is treated as an interval containing just that number.

```
friend: logical operator>= (
    SPAinterval const& i1,  // interval
    SPAinterval const& i2   // interval
    );
```

Arithmetic comparison. The meaning in each case is that every number in the first interval bears the given relationship (greater than or equal to) with every number in the second, where a double is treated as an interval containing just that number.

SPAmatrix

Class: Mathematics

Purpose: Defines a 3x3 affine transformation acting on vectors and positions.

Derivation: SPAmatrix : –

SAT Identifier: None

Filename: base/baseutil/vector/matrix.hxx

Description: This class defines a 3x3 Euclidean affine transformation acting on vectors and positions. It is not a tensor.

Limitations: None

References: by BASE SPAtransf

Data:

None

Constructor:

```
public: SPAmatrix::SPAmatrix ();
```

C++ allocation constructor requests memory for this object but does not populate it.

```
public: SPAmatrix::SPAmatrix (
    SPAvector const&,       // first vector
    SPAvector const&,       // second vector
    SPAvector const&        // third vector
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

```
public: void* operator SPAmatrix::new (
    size_t alloc_size       // size of requested
                            // memory block
    );
```

C++ constructor.

```
public: void* operator SPAmatrix::new (
    size_t alloc_size,      // size of requested
                            // memory block
    AcisMemType alloc_type, // eDefault, or
                            // eSession, or
                            // eDocument, or
                            // eTemporary
    const char*  alloc_file,// name of file in
                            // which new occurred
    int alloc_line,         // line of file in
                            // which new occurred
    int* alloc_file_index   // must always be
                            // &alloc_file_index
    );
```

New operator for single instances on older compilers where overloading of new and delete are not permitted, and the MMGR_FREELIST compiler flag is given.

```
public: void* operator new SPAmatrix::[] (
    size_t alloc_size       // size of requested
                            // memory block
    );
```

New operator for arrays of instances on older compilers where overloading of new and delete are not permitted, and the MMGR_FREELIST compiler flag is given.

Destructor:

```
public: void operator SPAmatrix::delete (
    void*  alloc_ptr        // pointer to memory
                            // block to delete
    );
```

Delete operator for single instances on older compilers where overloading of new and delete are not permitted, and the MMGR_FREELIST compiler flag is given.

```
public: void operator delete SPAmatrix::[] (
    void*  alloc_ptr        // pointer to memory
                            // block to delete
    );
```

Delete operator for arrays of instances.

Methods:

```
public: SPAvector SPAmatrix::column (
    int in_col              // column number
    ) const;
```

Extract a column from a matrix.

```
public: void SPAmatrix::debug (
    char const*,            // title
    FILE*                   // file name
        = debug_file_ptr
    ) const;
```

Writes output about the matrix to the debug file or to the specified file.

```
public: void SPAposition::debug_str (
    char* str               // string
    ) const;
```

Gives the details of a position.

```
public: double SPAmatrix::determinant () const;
```

Returns the determinant of the matrix.

```
public: double SPAmatrix::element (
    int row,                // row value
    int col                 // column value
    ) const;
```

Extracts an element of the matrix.

```
public: SPAmatrix SPAmatrix::inverse () const;
```

Returns the inverse of a matrix. The most common case is for the matrix to represent a rotation matrix for a transform. In this case, the matrix will be orthogonal, with unit determinant.

```
public: logical SPAmatrix::is_identity () const;
```

Returns TRUE if a matrix is the identity matrix.

```
public: void* operator SPAmatrix::new[] (
    size_t alloc_size,      // size of requested
                            // memory block
    AcisMemType alloc_type, // eDefault, or
                            // eSession, or
                            // eDocument, or
                            // eTemporary
    const char* alloc_file, // name of file in
                            // which new occurred
    int alloc_line,         // line of file in
                            // which new occurred
    int* alloc_file_index   // must always be
                            // &alloc_file_index
    );
```

New operator for arrays of instances, with decorations to keep track of the file and line where the new was issued. This version is available only on newer compilers that permit overloading of new and delete, and when the MMGR_ENABLED compiler flag is given.

```
public: SPAmatrix const& SPAmatrix::operator*= (
    double const&           // double
    );
```

Multiplies a matrix by a double.

```
public: SPAmatrix const& SPAmatrix::operator*= (
    SPAmatrix const&        // matrix.
    );
```

Multiplies two matrices.

```
public: SPAmatrix const& SPAmatrix::operator*= (
    SPAtransf const&        // transformation
    );
```

Transforms a matrix, i.e., by an affine transformation.

```
public: SPAvector SPAmatrix::row (
    int in_row              // row number
    ) const;
```

Extracts a row from the matrix.

```
public: void SPAmatrix::set_element (
    int row,                // row number
    int col,                // column number
    double new_e            // element value
    );
```

Specifies the number of rows and columns should be used in the definition of a matrix.

```
public: SPAmatrix SPAmatrix::transpose () const;
```

Returns a transpose of the matrix.

```
friend: SPAmatrix operator* (
    double const&,          // # to multiply by
    SPAmatrix const&        // matrix
    );
```

MAC, NT, and UNIX platforms only. Multiplies a matrix by a double.

```
friend: SPAmatrix operator* (
    SPAmatrix const&,       // first matrix
    SPAmatrix const&        // second matrix
    );
```

MAC, NT, and UNIX platforms only. Multiplies two matrices.

```
friend: SPAmatrix operator* (
    SPAmatrix const&,       // matrix
    SPAtransf const&        // transform
    );
```

MAC, NT, and UNIX platforms only. Transforms a matrix by an affine transformation.

```
friend: SPAmatrix operator* (
    SPAmatrix const&,       // matrix
    SPAtransf const*        // transform
    );
```

MAC, NT, and UNIX platforms only. Transforms a matrix by an affine transformation.

```
friend: SPAposition operator* (
    SPAmatrix const&,       // matrix
    SPAposition const&      // position
    );
```

MAC, NT, and UNIX platforms only. Transforms a position by a matrix.

```
friend: SPAposition operator* (
    SPAposition const&,     // position
    SPAmatrix const&        // matrix
    );
```

MAC, NT, and UNIX platforms only. Transforms a position by a matrix.

```
friend: SPAvector operator* (
    SPAmatrix const&,       // matrix
    SPAvector const&        // vector
    );
```

MAC, NT, and UNIX platforms only. Transforms a matrix by an affine transformation.

```
friend: SPAvector operator* (
    SPAvector const&,       // vector
    SPAmatrix const&        // matrix
    );
```

MAC, NT, and UNIX platforms only. Transforms a vector by a matrix.

Related Fncs:

same_matrix, scaling

message_module

Class: Error Handling

Purpose: Contains all messages for a module.

Derivation: message_module : ACIS_OBJECT : –

SAT Identifier: None

Filename: base/baseutil/errorsys/errmsg.hxx

Description: Refer to purpose.

Limitations: None

References: None

Data:

None

Constructor:

```
public: message_module::message_module (
    char const*,            // name of module
    message_list const*     // message
    );
```

C++ constructor, creating a message_module using the specified parameters.

Destructor:

```
public: message_module::~message_module ();
```

C++ destructor, deleting a message_module.

Methods:

```
public: void message_module::load ();
```

This function loads the message module.

```
public: static void message_module::loadAll ();
```

This static function initiates load of all message_modules. Loading is handled by a message_loader which interfaces to an external database. This function does nothing if the message_module is already loaded.

```
public: message_list const* message_module::message (
    int                     // Offset into module's
                            // message list
    ) const;
```

Returns a pointer to the message data for an offset into a module's message list

```
public:char** message_module::messageForUpdate (
    int idx                 // list index
    );
```

Returns the address of the char* pointer so the caller can assign their own string into the message_list.

```
public: err_mess_type message_module::message_code (
    int                     // Offset into module's
                            // message list
    ) const;
```

Generates an error number from an offset into a module's message list. Build tools generate a header file containing symbolic definitions of error numbers using this method.

```
public:int message_module::message_index ();;
```

Returns the message module index.

```
public: char const* message_module::module () const;
```

Returns the module's name.

```
public: message_module*
    message_module::next_message_module ();
```

Returns the next message module.

```
public:void message_module::unload ();;
```

Remove this module from the linked list. If there is no previous module in the list, then the element being removed is the head element.

```
public: static void message_module::unloadAll ();
```

This static function initiates unload of all message_modules. Unloading is handled by a message_loader which interfaces to an external database. This may be used to release memory resources used by the strings.

Related Fncs:

None

SPAnvector

Class: Mathematics

Purpose: Implements an *n* dimensional vector.

Derivation: SPAnvector : ACIS_OBJECT : –

SAT Identifier: None

Filename: base/baseutil/vector/complex.hxx

Description: Refer to Purpose.

Limitations: None

References: None

Data:

```
public double *values;
```
Array of values associated with the *n* dimensional vector.

```
public int size;
```
Number of items, *n*, in array of values associated with the *n* dimensional vector.

Constructor:

```
public: SPAnvector::SPAnvector ();
```

C++ constructor, creating a SPAnvector.

```
public: SPAnvector::SPAnvector (
    const SPAnvector&       // input vector
    );
```

C++ copy constructor, creating a SPAnvector. The array is whatever size the input SPAnvector is.

```
public: SPAnvector::SPAnvector (
    double                  // input double
    );
```

C++ constructor, creating a SPAnvector. The SPAnvector array is size 1.

```
public: SPAnvector::SPAnvector (
    double* in_values,      // input array
    int in_size             // size of array
    );
```

C++ constructor, creating a SPAnvector. The SPAnvector array is size in_size.

```
public: SPAnvector::SPAnvector (
    SPApar_pos              // input par pos
    );
```

C++ constructor, creating a SPAnvector. The SPAnvector array is size 2.

```
public: SPAnvector::SPAnvector (
    SPAposition             // input position
    );
```

C++ constructor, creating a SPAnvector. The SPAnvector array is size 3.

```
public: SPAnvector::SPAnvector (
    SPAvector               // input vector
    );
```

C++ constructor, creating a SPAnvector. The SPAnvector array is size 3.

Destructor:

```
public: SPAnvector::~SPAnvector ();
```

This should not be called directly. Use remove instead.

Methods:

```
public: double SPAnvector::length ();
```

Calculates the length of the SPAnvector by taking the square root of the sum of the square of each element in the array.

```
public: double SPAnvector::max_coord ();
```

Finds the maximum value in the array.

```
public: SPAnvector SPAnvector::norm ();
```

Finds the SPAnvector that is normal to this SPAnvector.

```
public: double SPAnvector::operator* (
    SPAnvector nv           // element to concatenate
    );
```

Concatenates the given SPAnvector to this SPAnvector. It fills in the SPAnvector if they are is not the right size. The respective elements are multiplied with one another.

```
public: SPAnvector SPAnvector::operator+ (
    SPAnvector nv           // given vector
    );
```

Adds the given SPAnvector to this SPAnvector. It fills in the SPAnvector if they are not the same size. The respective elements are added with one another.

```
public: SPAnvector SPAnvector::operator– (
    SPAnvector nv           // given vector
    );
```

Subtracts the given SPAnvector to this SPAnvector. It fills in the SPAnvector if they are not the same size. The respective elements are subtracted from one another.

```
public: SPAnvector SPAnvector::operator= (
    const SPAnvector&       // given vector
    );
```

Assigns the given SPAnvector values to the current vector. It fills in the SPAnvector if they are not the same size.

```
public: SPAnvector SPAnvector::scale (
    double s                // amount to scale
    );
```

Scales the SPAnvector by the given amount.

```
public: void SPAnvector::set (
    int in_size             // given size
    );
```

Changes the size of SPAnvector to be the given size.

Related Fncs:

None

option_header

Class: Modeler Control

Purpose: Records a value that denotes whether the option is *on*, *off*, or set to a given value.

Derivation: option_header : –

SAT Identifier: None

Filename: base/baseutil/option/option.hxx

Description: The class has a constructor that links an instance of it into a global chain that can then be inspected for reporting or changing the value. It maintains the default option value, as well as a stack of option values. Presently, the option chain is set up at initialization. New options can easily be added by making further static declarations of this class.

Limitations: None

References: None

Data:

None

Constructor:

```
public: void* operator option_header::new (
    size_t alloc_size       // object size in bytes
    );
```

Overloads the C++ new operator to allocate space on the portion of the heap controlled by ACIS. The C++ sizeof function can be used to obtain the size_t of the object.

```
public: void* operator option_header::new (
    size_t alloc_size,      // size of requested
                            // memory block
    AcisMemType alloc_type, // eDefault, or
                            // eSession, or
                            // eDocument, or
                            // eTemporary
    const char*  alloc_file,// name of file in
                            // which new occurred
    int alloc_line,         // line of file in
                            // which new occurred
    int* alloc_file_index   // must always be
                            // &alloc_file_index
    );
```

Overloads the C++ new operator to allocate space on the portion of the heap controlled by ACIS. The C++ sizeof function can be used to obtain the size_t of the object.

```
public: option_header::option_header (
    char const*,            // name
    char const*             // initial string
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

Creates an option_header with the specified name and initial string value and links it into the option list.

```
public: option_header::option_header (
    char const*,            // name
    double                  // initial value
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

Creates an option_header with the specified name and initial double value and links it into the option list.

```
public: option_header::option_header (
    char const*,            // name
    int                     // initial value
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

Creates an option_header with the specified name and initial integer value and links it into the option list.

```
public: void* operator new option_header::[](
    size_t alloc_size       // size of requested
                            // memory block
    );
```

New operator for arrays of instances on older compilers where overloading of new and delete are not permitted, and the MMGR_FREELIST compiler flag is given.

Destructor:

```
public: option_header::~option_header ();
```

C++ destructor, deleting an option_header. This destructor returns the option name to free storage, cleans up stacked values, and removes the header from the list.

```
public: void operator option_header::delete (
    void*  alloc_ptr        // pointer to memory
                            // block to delete
    );
```

Delete operator for single instances on older compilers where overloading of new and delete are not permitted, and the MMGR_FREELIST compiler flag is given.

```
public: void operator delete option_header::[](
    void*  alloc_ptr        // pointer to memory
                            // block to delete
    );
```

Delete operator for arrays of instances.

Methods:

```
public: int option_header::count () const;
```

Returns the value of the option_header if the option is of type integer or logical.

```
public: void option_header::display (
    FILE* fp                // file name
    ) const;
```

Prints the data contained in the option_header to the specified file.

```
public: logical option_header::is_default ();
```

Checks the current value against the default value. Returns TRUE if they are same or FALSE otherwise.

```
public: char const* option_header::name () const;
```

Returns the name of the option_header.

```
public: void* operator option_header::new[] (
    size_t alloc_size,      // size of requested
                            // memory block
    AcisMemType alloc_type, // eDefault, or
                            // eSession, or
                            // eDocument, or
                            // eTemporary
    const char*  alloc_file,// name of file in
                            // which new occurred
    int alloc_line,         // line of file in
                            // which new occurred
    int* alloc_file_index   // must always be
                            // &alloc_file_index
    );
```

New operator for arrays of instances, with decorations to keep track of the file and line where the new was issued. This version is available only on newer compilers that permit overloading of new and delete, and when the MMGR_ENABLED compiler flag is given.

```
public: option_header* option_header::next () const;
```

Returns the next option_header.

```
public: logical option_header::on () const;
```

Returns whether the option_header is on or off if the option type is logical_option.

```
public: void option_header::pop ();
```

Pops the stack to the previous value.

```
public: void option_header::push (
    char const*             // new value
    );
```

Pushes the new value onto the stack.

```
public: void option_header::push (
    double                  // double value
    );
```

Pushes the new value onto the stack.

```
public: void option_header::push (
    int                     // integer value
    );
```

Pushes the new value onto the stack.

```
public: void option_header::push (
    option_value const&     // option value
    );
```

Pushes the new value onto the stack.

```
public: void option_header::reset ();
```

Resets the option_header to the default value.

```
public: void option_header::set (
    char const*             // option name
    );
```

Sets the value of the option_header if the type is string_option.

```
public: void option_header::set (
    double                  // double value
    );
```

Sets the value of the option_header if is a double_option type.

```
public: void option_header::set (
    int                     // count
    );
```

Sets the value of the option_header if is an integer_option type or logical_option type.

```
public: void option_header::set (
    option_value const&     // option value
    );
```

Sets the option_value for the option_header if the option_value is the appropriate type.

```
public: void option_header::set_to_default ();
```

Resets the option to its default value, leaving the stack intact.

```
public: char const* option_header::string () const;
```

Returns the string value of the option_header if the option type is string_option.

```
public: option_type option_header::type () const;
```

Returns the type of option_header.

The option types are logical_option, int_option, double_option, string_option, or unknown_option.

```
public: double option_header::value () const;
```

Returns the value of the option_header if the option type is double_option.

Related Fncs:

find_option, get_option_list

output_callback

Class: Callbacks

Purpose: Creates output callback standard output for ACIS.

Derivation: output_callback : toolkit_callback : –

SAT Identifier: None

Filename: base/baseutil/acisio/acoutput.hxx

Description: The output_callback implements standard input for ACIS. Any time a standard C output call is made within ACIS, that call is redirected through this callback class. At that point the platform-specific implementation of output_callback obtains the input from its natural input stream.

Limitations: None

References: None

Data:

```
protected FILE *fp;
```

Pointer to the output file.

Constructor:

```
public: output_callback::output_callback (
    FILE* out_fp            // output file pointer
    );
```

C++ constructor, creating an output_callback using the specified parameters. Remembers the file to which this callback relates.

Destructor:

```
protected: virtual
    output_callback::~output_callback ();
```

C++ destructor, deleting an output_callback.

Methods:

```
public: virtual int output_callback::flush ();
```

Called in response to fflush for the specified file.

```
public: virtual int output_callback::print_string (
    const char* str         // string to write
    );
```

Writes string to the output device.

```
public: virtual int output_callback::write_data (
    const void* data,       // items in list
    int size,               // output device size
    int nitems              // number of items
    );
```

Writes from array data, *n* items of size to the output device.

Related Fncs:

None

SPAparameter

Class: Mathematics

Purpose: Defines a curve parameter value.

Derivation: SPAparameter : –

SAT Identifier: None

Filename: base/baseutil/vector/param.hxx

Description: This class defines a curve parameter value. It is a floating-point number, but it is declared as a class entity for consistency. Parameter values are invariant under transformations.

Limitations: None

References: by BASE SPApar_pos, SPApar_vec

Data:

None

Constructor:

```
public: void* operator SPAparameter::new (
    size_t alloc_size       // size of requested
                            // memory block
    );
```

C++ constructor.

```
public: void* operator SPAparameter::new (
    size_t alloc_size,      // size of requested
                            // memory block
    AcisMemType alloc_type, // eDefault, or
                            // eSession, or
                            // eDocument, or
                            // eTemporary
    const char*  alloc_file,// name of file in
                            // which new occurred
    int alloc_line,         // line of file in
                            // which new occurred
    int* alloc_file_index   // must always be
                            // &alloc_file_index
    );
```

New operator for single instances on older compilers where overloading of new and delete are not permitted, and the MMGR_FREELIST compiler flag is given.

```
public: SPAparameter::SPAparameter ();
```

C++ allocation constructor requests memory for this object but does not populate it.

```
public: SPAparameter::SPAparameter (
    double p                // double
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

```
public: void* operator new SPAparameter::[] (
    size_t alloc_size       // size of requested
                            // memory block
    );
```

New operator for arrays of instances on older compilers where overloading of new and delete are not permitted, and the MMGR_FREELIST compiler flag is given.

Destructor:

```
public: void operator SPAparameter::delete(
    void* alloc_ptr         // pointer to memory
                            // block to delete
    );
```

Delete operator for single instances on older compilers where overloading of new and delete are not permitted, and the MMGR_FREELIST compiler flag is given.

```
public: void operator delete SPAparameter::[] (
    void*  alloc_ptr        // pointer to memory
                            // block to delete
    );
```

Delete operator for arrays of instances.

Methods:

```
public: void SPAparameter::debug (
    FILE* fp                // file name
        = debug_file_ptr
    ) const;
```

Outputs the details of a parameter to the debug file or to the specified file.

```
public: operator SPAparameter::double () const;
```

Returns a double from a parameter.

```
public: void* operator SPAparameter::new[] (
    size_t alloc_size,      // size of requested
                            // memory block
    AcisMemType alloc_type, // eDefault, or
                            // eSession, or
                            // eDocument, or
                            // eTemporary
    const char* alloc_file, // name of file in
                            // which new occurred
    int alloc_line,         // line of file in
                            // which new occurred
    int* alloc_file_index   // must always be
                            // &alloc_file_index
    );
```

New operator for arrays of instances, with decorations to keep track of the file and line where the new was issued. This version is available only on newer compilers that permit overloading of new and delete, and when the MMGR_ENABLED compiler flag is given.

```
public: SPAparameter SPAparameter::operator*= (
    double rhs              // double
    );
```

Converts a double for multiplication to the right-hand system.

```
public: SPAparameter SPAparameter::operator+= (
    double rhs              // double
    );
```

Converts a double for addition to the right-hand system.

```
public: SPAparameter SPAparameter::operator– ()
const;
```

Negates a parameter.

```
public: SPAparameter SPAparameter::operator–= (
    double rhs              // double
    );
```

Converts a double for subtraction to the right-hand system.

```
public: SPAparameter SPAparameter::operator/ = (
    double rhs              // double
    );
```

Converts a double for division to the right-hand system.

Related Fncs:

same_SPApar_pos

```
friend: double operator* (
    double d,               // double
    SPAparameter const& p   // parameter
    );
```

Multiplies a parameter and a double.

```
friend: double operator* (
    int i,                  // integer
    SPAparameter const& p   // parameter
    );
```

Multiplies a parameter and an integer.

```
friend: double operator* (
    SPAparameter const& p,  // parameter
    double d                // double
    );
```

Multiplies a parameter and a double.

```
friend: double operator* (
    SPAparameter const& p,  // parameter
    int i                   // integer
    );
```

Multiplies a parameter and an integer.

```
friend: double operator* (
    SPAparameter const& p1, // parameter 1
    SPAparameter const& p2  // parameter 2
    );
```

Multiplies two parameters.

```
friend: double operator+ (
    double d,               // double
    SPAparameter const& p   // parameter
    );
```

Adds a parameter and a double.

```
friend: double operator+ (
    SPAparameter const& p,  // parameter
    double d                // double
    );
```

Adds a parameter and a double.

```
friend: double operator+ (
    SPAparameter const& p1, // parameter 1
    SPAparameter const& p2  // parameter 2
    );
```

Adds two parameters.

```
friend: double operator– (
    double d,               // double
    SPAparameter const& p   // parameter
    );
```

Subtracts a parameter from a double.

```
friend: double operator– (
    SPAparameter const& p,  // parameter
    double d                // double
    );
```

Subtracts a double from a parameter.

```
friend: double operator– (
    SPAparameter const& p1, // parameter 1
    SPAparameter const& p2  // parameter 2
    );
```

Subtracts the second parameter from the first parameter.

```
friend: double operator/ (
    double d,               // double
    SPAparameter const& p   // parameter
    );
```

Divides a double by a parameter.

```
friend: double operator/ (
    SPAparameter const& p,  // parameter
    double d                // double
    );
```

Divides a parameter by a double.

```
friend: double operator/ (
    SPAparameter const& p1, // parameter 1
    SPAparameter const& p2  // parameter 2
    );
```

Divides the first parameter by the second parameter.

```
friend: logical operator< (
    double d,               // double
    SPAparameter const& p   // parameter
    );
```

Determines if the double is less than the parameter.

```
friend: logical operator< (
    SPAparameter const& p,  // parameter
    double d                // double
    );
```

Determines if the parameter is less than the double.

```
friend: logical operator< (
    SPAparameter const& p1, // parameter 1
    SPAparameter const& p2  // parameter 2
    );
```

Determines if the first parameter is less than the second parameter.

```
friend: logical operator<= (
    double d,               // double
    SPAparameter const& p   // parameter
    );
```

Determines if the double is less than or equal to the parameter.

```
friend: logical operator<= (
    SPAparameter const& p,  // parameter
    double d                // double
    );
```

Determines if the parameter is less than or equal to the double.

```
friend: logical operator<= (
    SPAparameter const& p1, // parameter 1
    SPAparameter const& p2  // parameter 2
    );
```

Determines if the first parameter is less than or equal to the second parameter.

```
friend: logical operator> (
    double d,               // double
    SPAparameter const& p   // parameter
    );
```

Determines if the double is greater than the parameter.

```
friend: logical operator> (
    SPAparameter const& p,  // parameter
    double d                // double
    );
```

Determines if the parameter is greater than the double.

```
friend: logical operator> (
    SPAparameter const& p1, // parameter 1
    SPAparameter const& p2  // parameter 2
    );
```

Determines if the first parameter is greater than the second parameter.

```
friend: logical operator>= (
    double d,               // double
    SPAparameter const& p   // parameter
    );
```

Determines if the double is greater than or equal to the parameter.

```
friend: logical operator>= (
    SPAparameter const& p,  // parameter
    double d                // double
    );
```

Determines if the parameter is greater than or equal to the double.

```
friend: logical operator>= (
    SPAparameter const& p1, // parameter 1
    SPAparameter const& p2  // parameter 2
    );
```

Determines if the first parameter is greater than or equal to the second parameter.

SPApar_box

Class: Mathematics

Purpose: Defines a bounding box in parameter space by four values of class parameter: low_u, high_u, low_v, high_v.

Derivation: SPApar_box : –

SAT Identifier: None

Filename: base/baseutil/vector/param.hxx

Description: The SPApar_box class defines a 2D bounding box in parameter space by four values of class parameter: low_u, high_u, low_v, high_v.

Limitations: None

References: BASE SPAinterval

Data:

None

Constructor:

```
public: void* operator SPApar_box::new (
    size_t alloc_size       // size of requested
                            // memory block
    );
```

C++ constructor.

```
public: SPApar_box::SPApar_box ();
```

C++ allocation constructor requests memory for this object but does not populate it.

```
public: SPApar_box::SPApar_box (
    SPApar_box const&       // parameter box
    );
```

C++ copy constructor requests memory for this object and populates it with the data from the object supplied as an argument.

```
public: SPApar_box::SPApar_box (
    SPAinterval const&,     // u interval
    SPAinterval const&      // v interval
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

Creates a SPApar_box from the *u* interval and the *v* interval. If either interval is empty, the SPApar_box is empty.

```
public: SPApar_box::SPApar_box (
    SPApar_pos const&       // parameter position
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

Creates a SPApar_box from a position.

```
public: SPApar_box::SPApar_box (
    SPApar_pos const&,      // 1st parameter position
    SPApar_pos const&       // 2nd parameter position
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

Creates a SPApar_box from two positions.

```
public: void* operator new SPApar_box::[] (
    size_t alloc_size       // size of requested
                            // memory block
    );
```

New operator for arrays of instances on older compilers where overloading of new and delete are not permitted, and the MMGR_FREELIST compiler flag is given.

Destructor:

```
public: void operator SPApar_box::delete (
    void* alloc_ptr         // pointer to memory
                            // block to delete
    );
```

Delete operator for single instances on older compilers where overloading of new and delete are not permitted, and the MMGR_FREELIST compiler flag is given.

```
public: void operator delete SPApar_box::[] (
    void*  alloc_ptr        // pointer to memory
                            // block to delete
    );
```

Delete operator for arrays of instances.

Methods:

```
public: logical SPApar_box::bounded () const;
```

Returns TRUE if the coordinate ranges are finite or FALSE otherwise.

```
public: logical SPApar_box::bounded_above () const;
```

Returns TRUE if the coordinate ranges are either finite or finite above or FALSE otherwise.

```
public: logical SPApar_box::bounded_below () const;
```

Returns TRUE if the coordinate ranges are either finite or finite below or FALSE otherwise.

```
public: void SPApar_box::debug (
    char const*,            // title
    FILE*                   // file name
        = debug_file_ptr
    ) const;
```

Prints a titles and debug information about SPApar_box to the debug file or to the specified file.

```
public: logical SPApar_box::empty () const;
```

Tests if the box is empty.

```
public: logical SPApar_box::finite () const;
```

Returns TRUE if the coordinate ranges are finite or FALSE otherwise.

```
public: logical SPApar_box::finite_above () const;
```

Returns TRUE if the coordinate ranges are finite above or FALSE otherwise.

```
public: logical SPApar_box::finite_below () const;
```

Returns TRUE if the coordinate ranges are finite below or FALSE otherwise.

```
public: logical SPApar_box::infinite () const;
```

Returns TRUE if any of the coordinate ranges is infinite or FALSE otherwise.

```
public: SPApar_pos SPApar_box::high () const;
```

Extracts the high end of the leading diagonal from the SPApar_box.

```
public: SPApar_pos SPApar_box::low () const;
```

Extracts the low end of the leading diagonal from the SPApar_box.

```
public: SPApar_pos SPApar_box::mid () const;
```

Extracts the middle of the leading diagonal from the SPApar_box.

```
public: void* operator SPApar_box::new (
    size_t alloc_size,      // size of requested
                            // memory block
    AcisMemType alloc_type, // eDefault, or
                            // eSession, or
                            // eDocument, or
                            // eTemporary
    const char* alloc_file, // name of file in
                            // which new occurred
    int alloc_line,         // line of file in
                            // which new occurred
    int* alloc_file_index   // must always be
                            // &alloc_file_index
    );
```

New operator for arrays of instances, with decorations to keep track of the file and line where the new was issued. This version is available only on newer compilers that permit overloading of new and delete, and when the MMGR_ENABLED compiler flag is given.

```
public: void* operator SPApar_box::new [](
    size_t alloc_size,      // size of requested
                            // memory block
    AcisMemType alloc_type, // eDefault, or
                            // eSession, or
                            // eDocument, or
                            // eTemporary
    const char* alloc_file, // name of file in
                            // which new occurred
    int alloc_line,         // line of file in
                            // which new occurred
    int* alloc_file_index   // must always be
                            // &alloc_file_index
    );
```

New operator for arrays of instances, with decorations to keep track of the file and line where the new was issued. This version is available only on newer compilers that permit overloading of new and delete, and when the MMGR_ENABLED compiler flag is given.

```
public: SPApar_box& SPApar_box::operator& = (
    SPApar_box const&       // parameter box
    );
```

Limits one box by another; i.e., this method forms the intersection of this box with the given box and returns the intersection box as the result.

```
public: SPApar_box& SPApar_box::operator+= (
    SPApar_vec const&       // parameter vector
    );
```

Translates a SPApar_box by a parameter box.

```
public: SPApar_box& SPApar_box::operator–= (
    SPApar_vec const&       // parameter vector
    );
```

Translates a SPApar_box.

```
public: logical SPApar_box::operator<< (
    SPApar_box const& b     // given box
    ) const;
```

Determines if the given box entirely encloses this box. This method returns TRUE if the given box is NULL; otherwise, it returns FALSE if this box is NULL. This method also returns TRUE if this box is strictly within the given box or is within the given box enlarged by SPAresabs in all four directions (+*u*, –*u*, +*v*, –*v*).

```
public: logical SPApar_box::operator>> (
    SPApar_box const&       // given box
    ) const;
```

Determines if this box entirely encloses given box. This method returns TRUE if this box is NULL; otherwise, it returns FALSE if given box is NULL. This method also returns TRUE if the given box is strictly within this box or is within this box enlarged by SPAresabs in all four directions (+*u*, –*u*, +*v*, –*v*).

```
public: logical SPApar_box::operator>> (
    SPApar_pos const&       // parameter position
    ) const;
```

Determines the parametric point containment. This method returns TRUE if the point is contained within this box or if this box is NULL. The point counts as within if it is strictly within the box or within the box enlarged by SPAresabs in all four directions (+*u*, –*u*, +*v*, –*v*).

```
public: SPApar_box& SPApar_box::operator| = (
    SPApar_box const&       // parameter box
    );
```

Compounds one box into another; i.e., this method extends this box until it also encloses the given box.

```
public: SPAinterval SPApar_box::u_range () const;
```

Extracts the constituent data from the SPApar_box as an interval in the *u* direction.

```
public: SPAinterval SPApar_box::v_range () const;
```

Extracts the constituent data from the SPApar_box as an interval in the *v* direction.

```
friend: SPAinterval operator% (
    SPApar_box const& b,     // parameter box
    SPApar_dir const& d      // parameter direction
    );
```

Finds the extent of a SPApar_box along a given direction.

```
friend: SPAinterval operator% (
    SPApar_dir const&,       // parameter direction
    SPApar_box const&        // parameter box
    );
```

Finds the extent of a SPApar_box along a given direction.

```
friend: logical operator&& (
    SPApar_box const&,       // first parameter box
    SPApar_box const&        // second parameter box
    );
```

Determines if two boxes overlap. This method returns TRUE if either box is NULL or if all the intervals of one box overlap the corresponding intervals of the other.

```
friend: SPApar_box operator& (
    SPApar_box const&,       // first parameter box
    SPApar_box const&        // second parameter box
    );
```

Finds the overlap of two boxes; i.e., this method finds the intersection of the two boxes.

```
friend: SPApar_box operator+ (
    SPApar_box const&,       // parameter box
    SPApar_vec const&        // parameter vector
    );
```

Translates a SPApar_box.

```
friend: SPApar_box operator+ (
    SPApar_vec const&,       // parameter vector
    SPApar_box const&        // parameter box
    );
```

Translates a SPApar_box.

```
friend: SPApar_box operator- (
    SPApar_box const&,       // parameter box
    SPApar_vec const&        // parameter vector
    );
```

Translates a SPApar_box.

```
friend: logical operator<< (
    SPApar_pos const& p,     // parameter position
    SPApar_box const& b      // parameter box
    );
```

Determines the parametric point containment. Returns TRUE if the point is contained within this box or if this box is NULL. The point counts as within if it is strictly within the box or within the box enlarged by SPAresabs in all four directions ($+u$, $-u$, $+v$, $-v$).

```
friend: SPApar_box operator| (
    SPApar_box const&,       // first parameter box
    SPApar_box const&        // second parameter box
    );
```

Combines two boxes into a box that encloses both parameter boxes.

```
public: logical SPApar_box::unbounded () const;
```

Returns TRUE if the coordinate ranges are infinite or FALSE otherwise.

```
public: logical SPApar_box::unbounded_above () const;
```

Returns TRUE if the coordinate ranges are either infinite or infinite above or FALSE otherwise.

```
public: logical SPApar_box::unbounded_below () const;
```

Returns TRUE if the coordinate ranges are either infinite or infinite below or FALSE otherwise.

Related Fncs:

same_SPApar_pos

SPApar_dir

Class: Mathematics

Purpose: Defines a direction vector (du,dv) in 2D parameter–space.

Derivation: SPApar_dir : SPApar_vec : –

SAT Identifier: None

Filename: base/baseutil/vector/param.hxx

Description: This class defines a parametric direction vector on a surface in 2D parameter–space.

Limitations: None

References: None

Data:

```
public SPAparameter du;
```
The vector in the u–direction.

```
public SPAparameter dv;
```
The vector in the v–direction.

Constructor:

```
public: SPApar_dir::SPApar_dir ();
```

Construct an un–initialized SPApar_dir.

```
public: SPApar_dir::SPApar_dir (
    double uval,            // u vector
    double vval             // v vector
    );
```

Construct and normalize a SPApar_dir from two doubles.

```
public: SPApar_dir::SPApar_dir (
    double uv[ 2 ]          // array of 2 vectors
    );
```

Construct and normalize a SPApar_dir from an array of two doubles.

```
public: SPApar_dir::SPApar_dir (
    SPApar_vec const&v      // parameter vector
                            // direction
    );
```

Construct and normalize a SPApar_dir from a SPApar_vec.

Destructor:

None

Methods:

None

Related Fncs:

same_par_pos

```
friend: SPApar_dir operator- (
    SPApar_dir const& u     // parameter direction
    );
```

Performs a unary minus operation.

```
friend: double operator% (
    SPApar_dir const& u,    // parameter direction
    SPApar_pos const& p     // parameter position
    );
```

Returns the scalar product between a parameter direction and parameter position.

```
friend: double operator% (
    SPApar_pos const& p,    // parameter position
    SPApar_dir const& u     // parameter direction
    );
```

Returns the scalar product between a parameter position and parameter direction.

SPApar_pos

Class: Mathematics

Purpose: Defines a parameter position in the parameter-space of a surface.

Derivation: SPApar_pos : –

SAT Identifier: None

Filename: base/baseutil/vector/param.hxx

Description: This class represents a 2D parameter value that defines a (*u*, *v*) parameter-space coordinate that, when evaluated on a surface, produces a 3D object space coordinate.

Limitations: None

References: BASE SPAparameter

Data:

```
public SPAparameter u;
```
The *u*-parameter.

```
public SPAparameter v;
```
The *v*-parameter.

Constructor:

```
public: void* operator SPApar_pos::new (
    size_t alloc_size       // size of requested
                            // memory block
    );
```

C++ constructor.

```
public: void* operator SPApar_pos::new (
    size_t alloc_size,      // size of requested
                            // memory block
    AcisMemType alloc_type, // persistence
    const char* alloc_file, // name of file in
                            // which new occurred
    int alloc_line,         // line of file in
                            // which new occurred
    int* alloc_file_index   // must always be
                            // &alloc_file_index
    );
```

New operator for arrays of instances, with decorations to keep track of the file and line where the new was issued. This version is available only on newer compilers that permit overloading of new and delete, and when the MMGR_ENABLED compiler flag is given.

```
public: SPApar_pos::SPApar_pos ();
```

C++ allocation constructor requests memory for this object but does not populate it.

```
public: SPApar_pos::SPApar_pos (
    double uval,            // u-parameter value
    double vval             // v-parameter value
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

```
public: SPApar_pos::SPApar_pos (
    double uv[ 2 ]          // array of two values
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

```
public: void* operator new SPApar_pos::[] (
    size_t alloc_size       // size of requested
                            // memory block
    );
```

New operator for arrays of instances on older compilers where overloading of new and delete are not permitted, and the MMGR_FREELIST compiler flag is given.

Destructor:

```
public: void operator SPApar_pos::delete (
    void* alloc_ptr         // pointer to memory
                            // block to delete
    );
```

Delete operator for single instances on older compilers where overloading of new and delete are not permitted, and the MMGR_FREELIST compiler flag is given.

```
public: void operator delete SPApar_pos::[] (
    void*  alloc_ptr        // pointer to memory
                            // block to delete
    );
```

Delete operator for arrays of instances.

Methods:

```
public: void SPApar_pos::debug (
    FILE* fp                // file name
        = debug_file_ptr
    ) const;
```

Outputs details of a SPApar_pos to the debug file or to the specified file.

```
public: void* operator SPApar_pos::new [](
    size_t alloc_size,      // size of requested
                            // memory block
    AcisMemType alloc_type, // eDefault, or
                            // eSession, or
                            // eDocument, or
                            // eTemporary
    const char* alloc_file, // name of file in
                            // which new occurred
    int alloc_line,         // line of file in
                            // which new occurred
    int* alloc_file_index   // must always be
                            // &alloc_file_index
    );
```

New operator for arrays of instances, with decorations to keep track of the file and line where the new was issued. This version is available only on newer compilers that permit overloading of new and delete, and when the MMGR_ENABLED compiler flag is given.

```
public: SPApar_pos const& SPApar_pos::operator*= (
    SPApar_transf const&    // parameter transform
    );
```

Transforms the SPApar_pos.

```
public: SPApar_pos const& SPApar_pos::operator+= (
    SPApar_vec const&       // parameter vector
    );
```

Adds a vector to a parameter position by offsetting a parameter position by a parameter vector.

```
public: SPApar_pos const& SPApar_pos::operator-= (
    SPApar_vec const&       // parameter vector
    );
```

Subtracts a vector from a parameter position.

```
friend: double operator% (
    SPApar_pos const&,      // parameter position
    SPApar_vec const&       // parameter vector
    );
```

Returns the scalar product of a position with a vector.

```
friend: double operator% (
    SPApar_vec const& v,    // parameter vector
    SPApar_pos const& p     // parameter position
    );
```

Returns the scalar product of a vector with a position.

```
friend: SPApar_pos operator+ (
    SPApar_pos const&,      // parameter position
    SPApar_vec const&       // parameter vector
    );
```

Returns the sum of a position with a vector by offsetting a parameter position by a parameter vector.

```
friend: SPApar_pos operator+ (
    SPApar_vec const& v,    // parameter vector
    SPApar_pos const& p     // parameter position
    );
```

Returns the sum of a vector with a position.

```
friend: SPApar_pos operator- (
    SPApar_pos const&,      // parameter position
    SPApar_vec const&       // parameter vector
    );
```

Returns the subtraction of a vector from a position.

```
friend: SPApar_vec operator- (
    SPApar_pos const&,      // 1st parameter position
    SPApar_pos const&       // 2nd parameter position
    );
```

Returns the subtraction of the first parameter position from the second parameter position by determining the displacement vector from the first parameter position to the second parameter position.

Related Fncs:

same_SPApar_pos

SPApar_transf

Class: Mathematics

Purpose: Defines a parameter space transformation containing scaling and translation components.

Derivation: SPApar_transf : –

SAT Identifier: None

Filename: base/baseutil/vector/param.hxx

Description: This class defines a parameter space transformation containing scaling and translation components. The SPAparameter, SPApar_pos, SPApar_vec, SPApar_dir, SPApar_box and SPApar_transf classes define a parameter value along a curve, a pair of parameter values for a parameter point on a surface, a parametric direction on a surface, and a 2d box in parameter space.

The SPApar_transf allows the manipulation of a SPApar_pos as follows:

```
SPApar_pos_new = SPApar_pos_old * SPApar_transf
```

where:

```
u_new = u_old * u_scale + du
v_new = v_old * v_scale + dv
```

Limitations: None

References: None

Data:

None

Constructor:

```
public: void* operator SPApar_transf::new (
    size_t alloc_size       // size of requested
                            // memory block
    );
```

C++ constructor.

```
public: SPApar_transf::SPApar_transf ();
```

C++ allocation constructor requests memory for this object but does not populate it.

```
public: SPApar_transf::SPApar_transf (
    double us,              // u scaling
    double vs,              // v scaling
    double u,               // u translation
    double v                // v translation
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

```
public: void* operator SPApar_transf::new (
    size_t alloc_size,      // size of requested
                            // memory block
    AcisMemType alloc_type, // eDefault, or
                            // eSession, or
                            // eDocument, or
                            // eTemporary
    const char* alloc_file, // name of file in
                            // which new occurred
    int alloc_line,         // line of file in
                            // which new occurred
    int* alloc_file_index   // must always be
                            // &alloc_file_index
    );
```

New operator for arrays of instances, with decorations to keep track of the file and line where the new was issued. This version is available only on newer compilers that permit overloading of new and delete, and when the MMGR_ENABLED compiler flag is given.

Destructor:

```
public: void operator SPApar_transf::delete(
    void* alloc_ptr          // pointer to memory
                             // block to delete
    );
```

Delete operator for single instances on older compilers where overloading of new and delete are not permitted, and the MMGR_FREELIST compiler flag is given.

```
public: void operator delete SPApar_transf::[] (
    void*  alloc_ptr         // pointer to memory
                             // block to delete
    );
```

Delete operator for arrays of instances.

Methods:

```
public: void SPApar_transf::debug (
    FILE* fp                 // file pointer
        = debug_file_ptr
    ) const;
```

Outputs to the specified debug file information about this class.

```
public: double SPApar_transf::delta_u () const;
```

Returns the translation change in *u*.

```
public: double SPApar_transf::delta_v () const;
```

Returns the translation change in *v*.

```
public: logical SPApar_transf::identity () const;
```

Returns whether or not the given transform is the identity transform.

```
public: void* operator SPApar_transf::new [] (
    size_t alloc_size,       // size of requested
                             // memory block
    AcisMemType alloc_type,  // eDefault, or
                             // eSession, or
                             // eDocument, or
                             // eTemporary
    const char* alloc_file,  // name of file in
                             // which new occurred
    int alloc_line,          // line of file in
                             // which new occurred
    int* alloc_file_index    // must always be
                             // &alloc_file_index
    );
```

New operator for arrays of instances, with decorations to keep track of the file and line where the new was issued. This version is available only on newer compilers that permit overloading of new and delete, and when the MMGR_ENABLED compiler flag is given.

```
public: void SPApar_transf::set_delta_u (
    double delta_u           // value to set
    );
```

Specifies the translation change in *u*.

```
public: void SPApar_transf::set_delta_v (
    double delta_v           // value to set
    );
```

Specifies the translation change in *v*.

```
public: void SPApar_transf::set_identity (
    logical i                // use identity
    );
```

Specifies whether or not to turn transform into identity transform.

```
public: void SPApar_transf::set_u_scale (
    double us                // u scale
    );
```

Specifies change in *u* scaling.

```
public: void SPApar_transf::set_v_scale (
    double vs               // v scale
    );
```

Specifies change in *v* scaling.

```
public: double SPApar_transf::u_scale () const;
```

Returns the value of the *u* scaling.

```
public: double SPApar_transf::v_scale () const;
```

Returns the value of the *v* scaling.

```
public: void* operator new SPApar_transf::[](
    size_t alloc_size       // size of requested
                            // memory block
    );
```

New operator for arrays of instances.

Related Fncs:

same_SPApar_pos

```
friend: SPApar_pos const& SPApar_pos::operator*= (
    SPApar_transf const&    // parameter transform
    );
```

Transform a SPApar_pos.

```
friend: SPApar_pos operator* (
    SPApar_pos const&,      // parameter position
    SPApar_transf const&    // parameter transform
    );
```

Transform a SPApar_pos.

SPApar_vec

Class: Mathematics

Purpose: Defines a vector (*du*, *dv*) in 2D parameter-space.

Derivation: SPApar_vec : –

SAT Identifier: None

Filename: base/baseutil/vector/param.hxx

Description: This class defines a vector (*du*, *dv*) in 2D parameter-space.

Limitations: None

References: BASE SPAparameter

Data:

```
public SPAparameter du;
```
The vector in the *u*-direction.

```
public SPAparameter dv;
```
The vector in the *v*-direction.

Constructor:

```
public: SPApar_vec::SPApar_vec ();
```

C++ allocation constructor requests memory for this object but does not populate it.

```
public: SPApar_vec::SPApar_vec (
    double u,               // u vector
    double v                // v vector
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

```
public: SPApar_vec::SPApar_vec (
    double uv[ 2 ]          // array of 2 vectors
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

```
public: SPApar_vec::SPApar_vec (
    SPApar_dir const&       // parameter direction
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

Creates a SPApar_vec from a SPApar_dir. This is defined inline after SPApar_dir has been defined.

```
public: void* operator SPApar_vec::new (
    size_t alloc_size       // size of requested
                            // memory block
    );
```

C++ constructor.

```
public: void* operator SPApar_vec::new (
    size_t alloc_size,      // size of requested
                            // memory block
    AcisMemType alloc_type, // eDefault, or
                            // eSession, or
                            // eDocument, or
                            // eTemporary
    const char* alloc_file, // name of file in
                            // which new occurred
    int alloc_line,         // line of file in
                            // which new occurred
    int* alloc_file_index   // must always be
                            // &alloc_file_index
    );
```

New operator for arrays of instances, with decorations to keep track of the file and line where the new was issued. This version is available only on newer compilers that permit overloading of new and delete, and when the MMGR_ENABLED compiler flag is given.

Destructor:

```
public: void operator SPApar_vec::delete (
    void* alloc_ptr         // pointer to memory
                            // block to delete
    );;
```

Delete operator for single instances on older compilers where overloading of new and delete are not permitted, and the MMGR_FREELIST compiler flag is given.

Methods:

```
public: void SPApar_vec::debug (
    FILE* fp                // file name
        = debug_file_ptr
    ) const;
```

Outputs details of a SPApar_vec to the debug file or to the specified file.

```
public: logical SPApar_vec::is_zero (
    const double tol        // Zero tolerance
        = SPAresabs         //default
    ) const;
```

Returns TRUE if a radius function is zero everywhere, to within a given tolerance; otherwise, it returns FALSE.

```
public: double SPApar_vec::len () const;
```

Returns the length of the SPApar_vec.

```
public: double SPApar_vec::len_sq () const;
```

Returns (du * du + dv * dv).

```
public: void* operator SPApar_vec::new [](
    size_t alloc_size,      // size of requested
                            // memory block
    AcisMemType alloc_type, // eDefault, or
                            // eSession, or
                            // eDocument, or
                            // eTemporary
    const char* alloc_file, // name of file in
                            // which new occurred
    int alloc_line,         // line of file in
                            // which new occurred
    int* alloc_file_index   // must always be
                            // &alloc_file_index
    );
```

New operator for arrays of instances, with decorations to keep track of the file and line where the new was issued. This version is available only on newer compilers that permit overloading of new and delete, and when the MMGR_ENABLED compiler flag is given.

```
public: SPApar_vec const& SPApar_vec::operator*= (
    double                  // double
    );
```

Multiples a parameter by a scalar.

```
public: SPApar_vec const& SPApar_vec::operator+= (
    SPApar_vec const&       // parameter vector
    );
```

Adds two parameter vectors.

```
public: SPApar_vec const& SPApar_vec::operator–= (
    SPApar_vec const&       // parameter vector
    );
```

Performs a binary minus operation.

```
public: SPApar_vec const& SPApar_vec::operator/ = (
    double                  // scalar value
    );
```

Divides a parameter vector by a scalar value.

```
friend: double operator% (
    SPApar_vec const&,      // 1st parameter vector
    SPApar_vec const&       // 2nd parameter vector
    );
```

Returns the dot product of two parameter vectors.

```
friend: double operator* (
    SPApar_vec const&,      // 1st parameter vector
    SPApar_vec const&       // 2nd parameter vector
    );
```

Returns the cross product of two parameter vectors.

```
friend: SPApar_vec operator* (
    double d,               // scalar value
    SPApar_vec const& v     // parameter vector
    );
```

Multiplies a parameter vector by a scalar value.

```
friend: SPApar_vec operator* (
    SPApar_vec const&,       // parameter vector
    double                   // scalar value
    );
```

Multiplies a parameter vector by a scalar value.

```
friend: SPApar_vec operator+ (
    SPApar_vec const&,       // 1st parameter vector
    SPApar_vec const&        // 2nd parameter vector
    );
```

Adds the two parameter vectors.

```
friend: SPApar_vec operator– (
    SPApar_vec const&        // parameter vector
    );
```

Performs a unary minus operation.

```
friend: SPApar_vec operator– (
    SPApar_vec const&,       // 1st parameter vector
    SPApar_vec const&        // 2nd parameter vector
    );
```

Performs a binary minus operation.

```
friend: SPApar_vec operator/ (
    SPApar_vec const&,       // parameter vector
    double                   // scalar value
    );
```

Divides a parameter vector by a scalar value.

```
public: void* operator new SPApar_vec::[] (
    size_t alloc_size        // size of requested
                             // memory block
    );
```

New operator for arrays of instances on older compilers where overloading of new and delete are not permitted, and the MMGR_FREELIST compiler flag is given.

```
public: void operator delete SPApar_vec::[] (
    void*  alloc_ptr         // pointer to memory
                             // block to delete
    );
```

Delete operator for arrays of instances.

Related Fncs:

same_SPApar_pos

SPAposition

Class: Mathematics

Purpose: Represents position vectors (points) in 3D Cartesian space that are subject to certain vector and transformation operations.

Derivation: SPAposition : –

SAT Identifier: None

Filename: base/baseutil/vector/position.hxx

Description: This class represents position vectors (points) in 3D Cartesian space that are subject to certain vector and transformation operations. This class is distinct from the vector class which is a displacement and is origin independent.

Limitations: None

References: None

Data:

None

Constructor:

```
public: inline SPAposition::SPAposition ();
```

C++ allocation constructor requests memory for this object but does not populate it.

```
public: inline SPAposition::SPAposition (
    double p[ 3 ]            // array of 3 doubles
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

Creates a SPAposition using the specified array of three doubles.

```
public: inline SPAposition::SPAposition (
    double xi,              // x–coordinate
    double yi,              // y–coordinate
    double zi               // z–coordinate
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

Creates a SPAposition using the *x,y,z* coordinates.

```
public: inline SPAposition::SPAposition (
    SPAposition const& p    // position
    );
```

C++ copy constructor requests memory for this object and populates it with the data from the object supplied as an argument.

```
public: void* operator SPAposition::new (
    size_t alloc_size       // size of requested
                            // memory block
    );
```

C++ constructor.

```
public: void* operator SPAposition::new (
    size_t alloc_size,      // size of requested
                            // memory block
    AcisMemType alloc_type, // eDefault, or
                            // eSession, or
                            // eDocument, or
                            // eTemporary
    const char* alloc_file, // name of file in
                            // which new occurred
    int alloc_line,         // line of file in
                            // which new occurred
    int* alloc_file_index   // must always be
                            // &alloc_file_index
    );
```

New operator for arrays of instances, with decorations to keep track of the file and line where the new was issued. This version is available only on newer compilers that permit overloading of new and delete, and when the MMGR_ENABLED compiler flag is given.

```
public: void* operator new SPAposition::[] (
    size_t alloc_size       // size of requested
                            // memory block
    );
```

New operator for arrays of instances on older compilers where overloading of new and delete are not permitted, and the MMGR_FREELIST compiler flag is given.

Destructor:

```
public: void operator SPAposition::delete (
    void* alloc_ptr         // pointer to memory
                            // block to delete
    );
```

Delete operator for single instances on older compilers where overloading of new and delete are not permitted, and the MMGR_FREELIST compiler flag is given.

```
public: void operator delete SPAposition::[] (
    void*  alloc_ptr        // pointer to memory
                            // block to delete
    );
```

Delete operator for arrays of instances.

Methods:

```
public: inline double& SPAposition::coordinate (
    int i                   // ith component
    );
```

Extracts the *i*th component value.

```
public: inline double SPAposition::coordinate (
    int i                   // ith component
    ) const;
```

Returns the *i*th component value.

```
public: void SPAposition::debug (
    FILE*                   // file name
        = debug_file_ptr
    ) const;
```

Writes information about the position to the debug file or the specified file.

```
public: void SPAposition::debug_str (
    Char* str               // string
    ) const;
```

Concatenates the information about the position to the passed string.

```
public: void* operator SPAposition::new [](
    size_t alloc_size,      // size of requested
                            // memory block
    AcisMemType alloc_type, // eDefault, or
                            // eSession, or
                            // eDocument, or
                            // eTemporary
    const char* alloc_file, // name of file in
                            // which new occurred
    int alloc_line,         // line of file in
                            // which new occurred
    int* alloc_file_index   // must always be
                            // &alloc_file_index
    );
```

New operator for arrays of instances, with decorations to keep track of the file and line where the new was issued. This version is available only on newer compilers that permit overloading of new and delete, and when the MMGR_ENABLED compiler flag is given.

```
public: SPAposition const& SPAposition::operator*= (
    SPAmatrix const&        // matrix
    );
```

Transforms a position.

```
public: SPAposition const& SPAposition::operator*= (
    SPAtransf const&        // transformation
    );
```

Transforms a position.

```
public: SPAposition const& SPAposition::operator+= (
    SPAvector const&        // vector
    );
```

Translates a position by a vector.

```
public: SPAposition const& SPAposition::operator-= (
    SPAvector const&        // vector
    );
```

Translates a position by a vector.

```
public: inline void SPAposition::set_coordinate (
    int i,                  // node position
    double new_c            // new value
    );
```

Sets the *i*th component value.

```
public: inline void SPAposition::set_x (
    double new_x            // x-coordinate
    );
```

Sets the *x*-coordinate value.

```
public: inline void SPAposition::set_y (
    double new_y            // y-coordinate
    );
```

Sets the *y*-coordinate value.

```
public: inline void SPAposition::set_z (
    double new_z            // z-coordinate
    );
```

Sets the *z*-coordinate value.

```
public: inline double& SPAposition::x ();
```

Extracts the *x*-coordinate value.

```
public: inline double SPAposition::x () const;
```

Returns the *x*-coordinate value.

```
public: inline double& SPAposition::y ();
```

Extracts the *y*-coordinate value.

```
public: inline double SPAposition::y () const;
```

Returns the *y*-coordinate value.

```
public: inline double& SPAposition::z ();
```

Extracts the *z*-coordinate value.

```
public: inline double SPAposition::z () const;
```

Returns the *z*-coordinate value.

```
friend: double operator% (
    SPAposition const&,      // position
    SPAvector const&         // vector
    );
```

Returns the scalar product of a position with a vector.

```
friend: double operator% (
    SPAvector const&,        // vector
    SPAposition const&       // position
    );
```

Returns the scalar product of a position with a vector.

```
friend: SPAposition operator* (
    SPAmatrix const&,        // matrix
    SPAposition const&       // position
    );
```

Transforms a position.

```
friend: SPAposition operator* (
    SPAposition const&,      // position
    double                   // double value
    );
```

Multiplies a position by a double.

```
friend: SPAposition operator* (
    SPAposition const&,      // position
    SPAmatrix const&         // matrix
    );
```

Transforms a position.

```
friend: SPAposition operator* (
    SPAposition const&,      // position
    SPAtransf const&         // transformation
    );
```

Transforms a position.

```
friend: SPAposition operator* (
    SPAposition const&,      // position
    SPAunit_vector const&    // vector
    );
```

Transforms a position using a vector.

```
friend: SPAposition operator* (
    SPAposition const& p,    // position
    SPAtransf const* t       // unit vector
    );
```

Returns the cross product of a position with a unit vector.

```
friend: SPAposition operator* (
    SPAunit_vector const&,  // unit vector
    SPAposition const&      // position
    );
```

Returns the cross product of a position with a unit vector.

```
friend: SPAposition operator+ (
    SPAposition const&,     // position
    SPAvector const&        // vector
    );
```

Translates a position by a vector.

```
friend: SPAposition operator+ (
    SPAvector const&,       // vector
    SPAposition const&      // position
    );
```

Translates a position by a vector.

```
friend: SPAposition operator– (
    SPAposition const&,     // position
    SPAvector const&        // vector
    );
```

Translates a position by a vector.

```
friend: SPAvector operator– (
    SPAposition const&,     // position
    SPAposition const&      // position
    );
```

Returns the displacement (i.e., a vector) as difference of two positions.

```
friend: logical same_point (
    SPAposition const& p1,  // position 1
    SPAposition const& p2,  // position 2
    const double res        // resolution
        = SPAresabs
    );
```

Returns TRUE if the two positions are same i.e., lie within the specified resolution or “SPAresabs” or FALSE otherwise.

Related Fncs:

None

toolkit_callback

Class: Scheme Interface, Callbacks

Purpose: Defines the toolkit_callback base calls and the toolkit_callback_list class.

Derivation: toolkit_callback : –

SAT Identifier: None

Filename: base/baseutil/geomhusk/tlkit_cb.hxx

Description: The toolkit_callback class is a base class from which callback lists are derived. It defines the toolkit_callback base class from which the actual callback classes are derived, and the toolkit_callback_list class that tracks the list of callbacks.

This file defines the base class used to track the callback lists. Generally, this class is not made available to the users; instead, they derive new classes from specific kinds of callback classes that are derived from the classes defined here and overloads the virtual execute method of the derived class.

Limitations: None

References: by BASE toolkit_callback_list

Data:

None

Constructor:

```
public: toolkit_callback::toolkit_callback ();
```

C++ constructor, creating a toolkit_callback.

```
public: void* operator toolkit_callback::new (
    size_t alloc_size       // size of requested
                            // memory block
    );
```

C++ constructor.

```
public: void* operator toolkit_callback::new (
    size_t alloc_size,      // size of requested
                            // memory block
    AcisMemType alloc_type, // eDefault, or
                            // eSession, or
                            // eDocument, or
                            // eTemporary
    const char* alloc_file, // name of file in
                            // which new occurred
    int alloc_line,         // line of file in
                            // which new occurred
    int* alloc_file_index   // must always be
                            // &alloc_file_index
    );
```

New operator for arrays of instances, with decorations to keep track of the file and line where the new was issued. This version is available only on newer compilers that permit overloading of new and delete, and when the MMGR_ENABLED compiler flag is given.

```
public: void* operator new toolkit_callback::[] (
    size_t alloc_size       // size of requested
                            // memory block
    );
```

New operator for arrays of instances on older compilers where overloading of new and delete are not permitted, and the MMGR_FREELIST compiler flag is given.

Destructor:

```
protected: virtual toolkit_callback::
~toolkit_callback ();
```

C++ destructor, deleting a toolkit_callback.

```
public: void operator toolkit_callback::delete (
    void* alloc_ptr         // pointer to memory
                            // block to delete
    );;
```

Delete operator for single instances on older compilers where overloading of new and delete are not permitted, and the MMGR_FREELIST compiler flag is given.

```
public: void operator delete toolkit_callback::[] (
    void*  alloc_ptr        // pointer to memory
                            // block to delete
    );
```

Delete operator for arrays of instances.

Methods:

```
public: void* operator toolkit_callback::new [](
    size_t alloc_size,      // size of requested
                            // memory block
    AcisMemType alloc_type, // eDefault, or
                            // eSession, or
                            // eDocument, or
                            // eTemporary
    const char* alloc_file, // name of file in
                            // which new occurred
    int alloc_line,         // line of file in
                            // which new occurred
    int* alloc_file_index   // must always be
                            // &alloc_file_index
    );
```

New operator for arrays of instances, with decorations to keep track of the file and line where the new was issued. This version is available only on newer compilers that permit overloading of new and delete, and when the MMGR_ENABLED compiler flag is given.

```
public: toolkit_callback* toolkit_callback::next ();
```

Specifies the next callback command.

```
public: toolkit_callback* toolkit_callback::prev ();
```

Specifies the previous callback command.

```
protected: void toolkit_callback::set_next (
    toolkit_callback* cb    // callback
    );
```

Sets the next toolkit_callback.

```
protected: void toolkit_callback::set_prev (
    toolkit_callback* cb     // callback
    );
```

Sets the previous toolkit_callback.

Related Fncs:

None

toolkit_callback_list

Class: Scheme Interface, Callbacks

Purpose: Stores the list of toolkit_callbacks.

Derivation: toolkit_callback_list : ACIS_OBJECT : –

SAT Identifier: None

Filename: base/baseutil/geomhusk/tlkit_cb.hxx

Description: The toolkit_callback_list class stores the list of toolkit_callbacks.

Limitations: None

References:

BASE	toolkit_callback
by BASE	toolkit_callback

Data:

None

Constructor:

```
public:
    toolkit_callback_list::toolkit_callback_list ();
```

C++ constructor, creating a toolkit_callback_list.

Destructor:

```
public: virtual
    toolkit_callback_list::~toolkit_callback_list();
```

C++ destructor, deleting a toolkit_callback_list.

Methods:

```
protected: void toolkit_callback_list::add (
    toolkit_callback*        // callback
    );
```

Adds a callback to the head of the list, so it will be the first callback called.

```
protected: void toolkit_callback_list::append (
    toolkit_callback*       // callback
    );
```

Appends a callback to the end of the list.

```
public: void toolkit_callback_list::clear ();
```

Removes and delete all callbacks.

```
public: toolkit_callback*
    toolkit_callback_list::first ();
```

Gets the first callback.

```
public: toolkit_callback*
    toolkit_callback_list::last ();
```

Gets the last callback, for routines that want to call in reverse order.

```
public: virtual void toolkit_callback_list::remove (
    toolkit_callback*       // callback
    );
```

Removes a callback from the callback list and deletes it. This means that you must not delete the callback yourself, and also that all callback objects must be allocated on the heap using ’new’.

```
public:void
    toolkit_callback_list::remove_no_dtor  (
    toolkit_callback*       // callback
    );
```

Removes a callback from the callback list.

```
public: void toolkit_callback_list::remove_no_dtor (
    toolkit_callback*       // callback
    );
```

Removes a callback from the callback list.

Related Fncs:

None

SPAtransf

Class: Construction Geometry, Transforms, Modifying Models

Purpose: Represents a general 3D affine transformation.

Derivation: SPAtransf : –

SAT Identifier: None

Filename: base/baseutil/vector/transf.hxx

Description: This class represents a general 3D affine transformation. It is a 4 X 3 matrix by which to multiply a homogeneous vector, but it is stored specially for efficiency.

Limitations: None

References: BASE SPAmatrix, SPAvector

Data:

None

Constructor:

```
public: SPAtransf::SPAtransf ();
```

C++ allocation constructor requests memory for this object but does not populate it.

```
public: SPAtransf::SPAtransf (
    SPAtransf const&        // transform
    );
```

C++ copy constructor requests memory for this object and populates it with the data from the object supplied as an argument.

```
public: void* operator SPAtransf::new (
    size_t alloc_size       // size of requested
                            // memory block
    );
```

C++ constructor.

```
public: void* operator SPAtransf::new (
    size_t alloc_size,      // size of requested
                            // memory block
    AcisMemType alloc_type, // eDefault, or
                            // eSession, or
                            // eDocument, or
                            // eTemporary
    const char*  alloc_file,// name of file in
                            // which new occurred
    int alloc_line,         // line of file in
                            // which new occurred
    int* alloc_file_index   // must always be
                            // &alloc_file_index
    );
```

New operator for single instances on older compilers where overloading of new and delete are not permitted, and the MMGR_FREELIST compiler flag is given.

```
public: void* operator new SPAtransf::[] (
    size_t alloc_size       // size of requested
                            // memory block
    );
```

New operator for arrays of instances on older compilers where overloading of new and delete are not permitted, and the MMGR_FREELIST compiler flag is given.

Destructor:

```
public: void operator SPAtransf::delete (
    void* alloc_ptr         // pointer to memory
                            // block to delete
    );
```

Delete operator for single instances on older compilers where overloading of new and delete are not permitted, and the MMGR_FREELIST compiler flag is given.

```
public: void operator delete SPAtransf::[] (
    void*  alloc_ptr        // pointer to memory
                            // block to delete
    );
```

Delete operator for arrays of instances.

Methods:

```
public: SPAmatrix SPAtransf::affine () const;
```

Returns the affine portion of the transformation. Always normalized: det == + or –1

```
public: logical SPAtransf::compose (
    const transf_decompose_data&// transformation
        data,                   // data
    logical rotate_xyz_axes     // rotate x, y, z
    );
```

Interprets the transf_decompose_data structure as a series of transformations:

```
[scalex] [scaley] [scalez] [shearxy] [shearxz]
[shearyz][rotatex] [rotatey] [rotatez] [translatex]
[translatey] [translatez]
```

Or, if the logical rotate_xyz_axes is FALSE, the sequence is:

```
[scalex] [scaley] [scalez] [shearxy] [shearxz]
[shearyz] [rotate_radians] [rotate_axis] [translatex]
[translatey] [translatez]
```

```
public: void SPAtransf::debug (
    char const*,            // leader string
    FILE*                   // pointer
        = debug_file_ptr
    ) const;
```

Outputs details of a transf to the specified file.

```
public: logical SPAtransf::decompose (
    transf_decompose_data& data // output data
    ) const;
```

Decomposes a non–degenerate transformation into data that represents a unique sequence of scaling, shearing, rotating and translating.

```
public: logical SPAtransf::identity () const;
```

If level is unspecified or 0, returns the type identifier transf_TYPE. If level is specified, returns transf_TYPE for that level of derivation from ENTITY. The level of this class is defined as transf_LEVEL.

```
public: SPAtransf SPAtransf::inverse () const;
```

Returns the inverse transformation. There must be no shear in the given transformation.

```
public: void* operator SPAtransf::new [](
    size_t alloc_size,       // size of requested
                             // memory block
    AcisMemType alloc_type,  // eDefault, or
                             // eSession, or
                             // eDocument, or
                             // eTemporary
    const char* alloc_file,  // name of file in
                             // which new occurred
    int alloc_line,          // line of file in
                             // which new occurred
    int* alloc_file_index    // must always be
                             // &alloc_file_index
    );
```

New operator for arrays of instances, with decorations to keep track of the file and line where the new was issued. This version is available only on newer compilers that permit overloading of new and delete, and when the MMGR_ENABLED compiler flag is given.

```
public: logical SPAtransf::operator!= (
    SPAtransf const& rhs     // transformation
    ) const;
```

Compares two transformations. This method does not allow any tolerance so it is not a general equality operator, but it it returns FALSE if one argument is a copy of the other.

```
public: SPAtransf const& SPAtransf::operator*= (
    SPAtransf const&         // transformation
    );
```

Multiplies two transformations.

```
public: logical SPAtransf::operator== (
    SPAtransf const&        // transformation
    ) const;
```

Compares two transformations. This method does not allow any tolerance so it is not a general equality operator, but it it returns TRUE if one argument is a copy of the other.

```
public: void SPAtransf::print () const;
```

Print transform data.

```
public: logical SPAtransf::reflect () const;
```

Determines if the transformation has a reflection component.

```
public: logical SPAtransf::rotate () const;
```

Determines if the transformation has a rotate component.

```
public: double SPAtransf::scaling () const;
```

Returns the scaling factor of transformation.

```
public: logical SPAtransf::shear () const;
```

Determines if the transformation has a shear component.

```
public: SPAvector SPAtransf::translation () const;
```

Return the translation portion of the transformation.

```
friend: SPAposition operator* (
    SPAposition const&,     // position
    SPAtransf const&        // transformation
    );
```

Transform a position.

```
friend: SPAtransf operator* (
    SPAtransf const&,       // first transform
    SPAtransf const&        // second transform
    );
```

Multiplies two transforms.

```
friend: SPAtransf operator* (
    SPAtransf const& t1,     // first transform
    SPAtransf const* t2      // second transform
    );
```

Multiplies two transforms.

```
friend: SPAunit_vector operator* (
    SPAunit_vector const&,   // unit vector
    SPAtransf const&         // transformation
    );
```

Transforms a unit vector. This method ignores the translation and scaling parts, but complains if there is a shear.

```
friend: SPAvector operator* (
    SPAvector const&,        // vector
    SPAtransf const&         // transformation
    );
```

Transforms a vector, ignoring the translation part of the transformation.

Internal Use: SPAtransf

Related Fncs: coordinate_transf, reflect_transf, rotate_transf, scale_transf, translate_transf

SPAunit_vector

Class: Mathematics

Purpose: Provides a direction in 3D Cartesian space that has unit length.

Derivation: SPAunit_vector : SPAvector : –

SAT Identifier: None

Filename: base/baseutil/vector/unitvec.hxx

Description: This class provides a direction in 3D Cartesian space that has unit length. Because it is a derived class of vector, it inherits the functionality of vectors. There are a few operations that are peculiar to unit vectors.

Limitations: None

References: None

Data:

None

Constructor:

```
public: SPAunit_vector::SPAunit_vector ();
```

C++ allocation constructor requests memory for this object but does not populate it.

```
public: SPAunit_vector::SPAunit_vector (
    double,                 // x-coordinate
    double,                 // y-coordinate
    double                  // z-coordinate
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

Creates a SPAunit_vector using the specified parameters. The result is normalized.

```
public: SPAunit_vector::SPAunit_vector (
    double u[ 3 ]           // x,y, & z values
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

Creates a SPAunit_vector using the specified parameters. The result is normalized.

Destructor:

None

Methods:

```
public: SPAunit_vector const&
SPAunit_vector::operator*= (
    SPAtransf const&        // transform
    );
```

Transforms a unit vector by the rotation matrix in a transformation. This method returns an error if the transformation contains a shear component.

```
friend: double operator% (
    SPAposition const&,      // position
    SPAunit_vector const&    // unit vector
    );
```

Returns the scalar product of a position and a SPAunit_vector. It is declared explicitly to avoid an ambiguity.

```
friend: double operator% (
    SPAunit_vector const&,   // unit vector
    SPAposition const&       // position
    );
```

Returns the scalar product of a position and a SPAunit_vector. It is declared explicitly to avoid an ambiguity.

```
friend: SPAposition operator* (
    SPAposition const&,      // position
    SPAunit_vector const&    // unit vector
    );
```

Returns a position as a cross–product of a unit vector with a position.

```
friend: SPAposition operator* (
    SPAunit_vector const&,   // unit vector
    SPAposition const&       // position
    );
```

Returns a position as a cross–product of a position with a unit vector.

```
friend: SPAunit_vector operator* (
    SPAunit_vector const&,   // unit vector
    SPAtransf const&         // transformation
    );
```

Transforms a unit vector by the rotation matrix in a transformation. This method returns an error if the transformation contains a shear component.

```
friend: SPAunit_vector operator* (
    SPAunit_vector const&,   // unit vector
    SPAtransf const*         // transformation
    );
```

Transforms a unit vector by the rotation matrix in a transformation. This method returns an error if the transformation contains a shear component.

```
friend: SPAunit_vector operator- (
    SPAunit_vector const&   // unit vector
    );
```

Performs a unary minus operation.

Related Fncs:

normalise

SPAvector

Class: Mathematics

Purpose: Represents a displacement vector in 3D Cartesian space.

Derivation: SPAvector : –

SAT Identifier: None

Filename: base/baseutil/vector/vector.hxx

Description: This class represents a displacement vector in 3D Cartesian space.

Limitations: None

References: by BASE SPAtransf

Data:

None

Constructor:

```
public: SPAvector::SPAvector ();
```

C++ allocation constructor requests memory for this object but does not populate it.

```
public: SPAvector::SPAvector (
    double v[ 3 ]           // array of 3 doubles
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

Creates a SPAvector from an array of three doubles representing the *x*, *y*, and *z* coordinate values.

```
public: SPAvector::SPAvector (
    double x,               // x–coordinate
    double y,               // y–coordinate
    double z                // z–coordinate
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

Creates a SPAvector from three doubles representing the *x*, *y*, and *z* coordinate values.

```
public: SPAvector::SPAvector (
    SPAvector const& v      // vector
    );
```

C++ copy constructor requests memory for this object and populates it with the data from the object supplied as an argument.

```
public: void* operator SPAvector::new (
    size_t alloc_size       // size of requested
                            // memory block
    );
```

C++ constructor.

```
public: void* operator SPAvector::new (
    size_t alloc_size,      // size of requested
                            // memory block
    AcisMemType alloc_type, // eDefault, or
                            // eSession, or
                            // eDocument, or
                            // eTemporary
    const char*  alloc_file,// name of file in
                            // which new occurred
    int alloc_line,         // line of file in
                            // which new occurred
    int* alloc_file_index   // must always be
                            // &alloc_file_index
    ) ;
```

New operator for single instances on older compilers where overloading of new and delete are not permitted, and the MMGR_FREELIST compiler flag is given.

```
public: void* operator new SPAvector::[] (
    size_t alloc_size       // size of requested
                            // memory block
    );
```

New operator for arrays of instances on older compilers where overloading of new and delete are not permitted, and the MMGR_FREELIST compiler flag is given.

Destructor:

```
public: void operator SPAvector::delete (
    void* alloc_ptr         // pointer to memory
                            // block to delete
    );
```

Delete operator for single instances on older compilers where overloading of new and delete are not permitted, and the MMGR_FREELIST compiler flag is given.

```
public: void operator delete SPAvector::[] (
    void*  alloc_ptr        // pointer to memory
                            // block to delete
    );
```

Delete operator for arrays of instances.

Methods:

```
public: double& SPAvector::component (
    int i                   // ith component
    );
```

Extracts the *i*th component of a vector and allows it to be modified.

```
public: double SPAvector::component (
    int i                   // ith component
    ) const;
```

Accesses the *i*th component. *i* must equal 0, 1, or 2.

```
public: void SPAvector::debug (
    FILE*                   // file pointer
        = debug_file_ptr
    ) const;
```

Outputs debug information to the screen or to the specified file.

```
public: void SPAvector::debug_str (
    char* str               // string
    ) const;
```

Concatenates the debug information to the passed string.

```
public: logical SPAvector::is_zero (
    const double tol        // zero tolerance
        = SPAresabs
    ) const;
```

Returns TRUE if a radius function is zero everywhere, to within a given tolerance; otherwise, it returns FALSE.

```
public: double SPAvector::len () const;
```

Returns the length of a vector.

```
public: double SPAvector::len_sq () const;
```

Returns the square of the length of the vector.

```
public: SPAvector SPAvector::make_ortho ();
```

Multiplies a vector by a scalar value.

```
public: double SPAvector::max_norm (
    int& i                  // max component
    ) const;
```

This gets the maximum of the fabs of each component, and which component was the maximum. In case of a “tie” the index i will default to the larger index. For example, for the vector (1,1,1), i = 2.

```
public: void* operator SPAvector::new [](
    size_t alloc_size,      // size of requested
                            // memory block
    AcisMemType alloc_type, // eDefault, or
                            // eSession, or
                            // eDocument, or
                            // eTemporary
    const char* alloc_file, // name of file in
                            // which new occurred
    int alloc_line,         // line of file in
                            // which new occurred
    int* alloc_file_index   // must always be
                            // &alloc_file_index
    );
```

New operator for arrays of instances, with decorations to keep track of the file and line where the new was issued. This version is available only on newer compilers that permit overloading of new and delete, and when the MMGR_ENABLED compiler flag is given.

```
public: double SPAvector::numerically_stable_len ()
const;
```

This method is more expensive than len(), but (theoretically) gives the same value and is stable for very small (those for which v%v would be lost in numerical noise) or very large norms (those for which v%v would give overflow). Not appropriate for general use.

```
public: SPAvector const& SPAvector::operator*= (
    double                  // scalar value
    );
```

Multiplication of a vector by a scalar.

```
public: SPAvector const& SPAvector::operator*= (
    SPAmatrix const&        // 3 x 3 matrix
    );
```

Transforms a vector by a 3 X 3 matrix.

```
public: SPAvector const& SPAvector::operator*= (
    SPAtransf const&        // transform
    );
```

Transforms a vector by an affine transformation.

```
public: SPAvector const& SPAvector::operator+= (
    SPAvector const&        // vector
    );
```

Adds two vectors.

```
public: SPAvector const& SPAvector::operator–= (
    SPAvector const&        // vector
    );
```

Binary minus operation.

```
public: SPAvector const& SPAvector::operator/ = (
    double                  // scalar value
    );
```

Divides a vector by a scalar value.

```
public: SPAunit_vector SPAvector::orthogonal ()
const;
```

This returns some SPAunit_vector which is orthogonal to the given one. If the given vector is less than SPAresmch in length, it returns the unit vector (0,0,1).

```
public: void SPAvector::set_component (
    int i,                  // ith component
    double new_c            // new value
    );
```

Sets the value of *i*th component of a vector.

```
public: void SPAvector::set_x (
    double new_x            // x–coordinate
    );
```

Sets the *x*-coordinate of a vector.

```
public: void SPAvector::set_y (
    double new_y            // y–coordinate
    );
```

Sets the *y*-coordinate of a vector.

```
public: void SPAvector::set_z (
    double new_z            // z–coordinate
    );
```

Sets the *z*-coordinate of a vector.

```
public: double& SPAvector::x ();
```

Extracts the *x*-component of a vector for an update and allows it to be modified.

```
public: double SPAvector::x () const;
```

Extracts the *x*-component of a vector in 3D Cartesian space.

```
public: double& SPAvector::y ();
```

Extracts the *y*-component of a vector for an update and allows it to be modified.

```
public: double SPAvector::y () const;
```

Extracts the *y*-component of a vector.

```
public: double& SPAvector::z ();
```

Extracts the *z*-component of a vector for an update and allows it to be modified.

```
public: double SPAvector::z () const;
```

Extracts the *z*-component of a vector.

Related Fncs:

antiparallel, biparallel, normalise, parallel, perpendicular, same_vector

```
friend: double operator% (
    SPAposition const&,      // position
    SPAvector const&         // vector
    );
```

Scalar product of a position and a vector.

```
friend: double operator% (
    SPAvector const&,        // vector
    SPAposition const&       // position
    );
```

Scalar product of a position and a vector.

```
friend: double operator% (
    SPAvector const&,        // first vector
    SPAvector const&         // second vector
    );
```

Scalar product of two vectors.

```
friend: SPAvector operator* (
    double,                  // scalar value
    SPAvector const&         // vector
    );
```

Multiplies a vector by a scalar value.

```
friend: SPAvector operator* (
    SPAmatrix const&,        // 3 x 3 matrix
    SPAvector const&         // vector
    );
```

Transforms a vector by a 3 X 3 matrix.

```
friend: SPAvector operator* (
    SPAvector const&,        // vector
    double                   // scalar value
    );
```

Multiplies a vector by a scalar value.

```
friend: SPAvector operator* (
    SPAvector const&,       // vector
    SPAmatrix const&        // 3 x 3 matrix
    );
```

Transforms a vector by a 3 X 3 matrix.

```
friend: SPAvector operator* (
    SPAvector const&,       // vector
    SPAtransf const&        // transform
    );
```

Transforms a vector by an affine transformation.

```
friend: SPAvector operator* (
    SPAvector const&,       // vector
    SPAtransf const*        // transform
    );
```

Transforms a vector by an affine transformation.

```
friend: SPAvector operator* (
    SPAvector const&,       // vector
    SPAvector const&        // vector
    );
```

Cross product of two vectors. Also applies to unit vectors.

```
friend: SPAvector operator+ (
    SPAvector const&,       // first vector
    SPAvector const&        // second vector
    );
```

Addition of two vectors.

```
friend: SPAvector operator– (
    SPAvector const&        // vector
    );
```

Unary minus operation.

```
friend: SPAvector operator– (
    SPAvector const&,        // first vector
    SPAvector const&         // second vector
    );
```

Binary minus operation.

```
friend: SPAvector operator/ (
    SPAvector const&,        // vector
    double                   // scalar value
    );
```

Division of a vector by a scalar.

VOID_LIST

Class: Entity

Purpose: Creates a variable-length list of void*'s.

Derivation: VOID_LIST : ACIS_OBJECT : –

SAT Identifier: None

Filename: base/baseutil/lists/vlists.hxx

Description: This class implements a variable-length list of void*'s. It provides a constructor (which creates an empty list) and destructor functions to add a void* (only if not already there), to look one up by pointer value, and to count the number of void*'s listed. It also provides an overloaded "[ ]" operator for access by position.

For best performance in loops that step through this list, have the loops increment rather than decrement the index counter. Internal operations for methods like operator[] and remove store the index counter from the previous operation allowing faster access to the next indexed item when indexing up.

The current implementation uses hashing so that look up is fast provided lists are not very long; it is also efficient for very short lists and for repeated lookups of the same void*.

Limitations: None

References: None

Data:

None

Constructor:

```
public: VOID_LIST::VOID_LIST ();
```

C++ constructor, creating a VOID_LIST.

```
public: VOID_LIST::VOID_LIST (
    VOID_LIST const&        // entity to copy
    );
```

Copy constructor which copy the whole list (complete with deleted entries, if any, so that the indices in the copy match those in the original).

Destructor:

```
public: VOID_LIST::~VOID_LIST ();
```

C++ destructor, deleting a VOID_LIST.

Methods:

```
public: int VOID_LIST::add (
    void*                   // void pointer
    );
```

Adds an item to the list if not already there, and always returns the index.

```
public: int VOID_LIST::byte_count (
    logical countSelf       // list or header
        = TRUE              // toggle
    ) const;
```

Return the size of the list.

```
public: void VOID_LIST::clear ();
```

Empties a list ready for construction of a new one.

```
public: int VOID_LIST::count () const;
```

Count how many item*'s there are in the list including deleted entries.

```
public: void VOID_LIST::init () const;
```

Return item*'s in list order, ignoring deleted items. Call init once, then next repeatedly until it returns NULL. Note that next returns the undeleted item most closely following the one most recently returned by next or operator [ ], except that if that value was NULL the value of next is undefined.

```
public: int VOID_LIST::iteration_count () const;
```

Counts how many entities there are in the list not including deleted entries. Uses the iterator.

```
public: int VOID_LIST::lookup (
    void const*             // item to find
    ) const;
```

Search for an item in the list. Return its index number, or –1 if it is not there.

```
public: void* VOID_LIST::next () const;
```

Return item*'s in list order, ignoring deleted items. Call init first time, then next repeatedly until it returns NULL. Note that next returns the undeleted item most closely following the one most recently returned by next or operator [ ], except that if that value was NULL the value of next is undefined.

```
public: VOID_LIST& VOID_LIST::operator= (
    VOID_LIST const&        // pointer
    ) ;
```

Explicit operator.

```
public: void* VOID_LIST::operator[] (
    int                     // index number
    ) const;
```

Returns the indexed item, or NULL if the index is out of range, or LIST_ENTRY_DELETED if the indexed entry has been deleted.

```
public: int VOID_LIST::remove (
    void const*             // pointer to item
    );
```

Delete an item from the list. This does not free space, and leaves a tombstone in the linear list which count and operator [ ] will notice. But lookup will not find this item, nor will init or next. The return value is the lookup value of the old entry.

Related Fncs:

None