Chapter 2.
How ACIS Uses C++

Topic: *C++ Interface

ACIS is written in C++ and consists of a set of C++ functions and classes (including data and member functions, or *methods*). A developer uses these functions and classes to create an end user 3D modeling application. This chapter describes how ACIS uses features of C++.

ACIS is a solid modeler, but wireframe and surface models may also be represented in ACIS. ACIS integrates wireframe, surface, and solid modeling by allowing these alternative model representations to coexist naturally in a unified data structure, which is implemented as a hierarchy of C++ classes.

ACIS utilizes key features of C++, such as encapsulation, overloading, etc. to create a modular and maintainable interface. The API function interface remains stable from release to release, while the class interface may change.

The key aspects of C++ that ACIS takes advantage of are:

- Data encapsulation
- Overloading class constructors
- Copying objects
- Overloading class methods and operators
- Overloading functions

Data Encapsulation

Topic: *C++ Interface

A class provides not only data storage, but also methods (or member functions) that can access and manipulate that data. The concept of having private data and public methods for accessing data is common to ACIS classes. This is how ACIS uses C++ classes to provide data encapsulation and modularization. This permits reorganization of the internals of any component while maintaining the interface.

Although it is possible to make the class data members accessible to anyone (public), it is much more common to hide the data members by making them private. Access to private class data members is restricted to the public class methods. Not all class methods need to be exposed, so some methods are also private.

For example, consider the SPAposition class, which is used to define a position in space. The declaration for this class is shown in Example 2-1. The internal data structure for this is an array of three real numbers and it has class methods for the *x*, *y*, and *z* components of the position: x, y, and z. At some later date, if the data structure for SPAposition were to use three separate real numbers instead of an array, this could be implemented and still be transparent to the users of this class, because the users can only access the public x, y, and z methods and not the private data elements directly.

Note *ACIS globally defines the keyword* real *to be equivalent to a C++* double *or* float *data type using a* #define *statement.*

C++ Example

```
class DECL_KERN SPAposition {
    real coord[ 3 ];   // x, y and z coordinates of the position

public:

    // Force creation of all positions to be by constructor.
    SPAposition() {}

    // Construct a position from three doubles.
    SPAposition( double xi, double yi, double zi ) {
        coord[ 0 ] = (real) xi;
        coord[ 1 ] = (real) yi;
        coord[ 2 ] = (real) zi;
    }

    // Construct a position from an array of three doubles.
    SPAposition( double p[ 3 ] ) {
        coord[ 0 ] = (real) p[ 0 ];
        coord[ 1 ] = (real) p[ 1 ];
        coord[ 2 ] = (real) p[ 2 ];
    }

    // Copy a position
    SPAposition( SPAposition const &p ) {
        coord[ 0 ] = p.coord[ 0 ];
        coord[ 1 ] = p.coord[ 1 ];
        coord[ 2 ] = p.coord[ 2 ];
    }

    // Extract a coordinate value.
    double x() const   { return coord[ 0 ]; }
    double y() const   { return coord[ 1 ]; }
    double z() const   { return coord[ 2 ]; }

    double coordinate( int i ) const { return coord[ i ]; }
    // Extract a coordinate value for update.
    double &x() { return coord[ 0 ]; }
    double &y() { return coord[ 1 ]; }
    double &z() { return coord[ 2 ]; }

    double &coordinate( int i ) { return coord[ i ]; }

    // Set coordinate values.
    void set_x( double new_x ) { coord[ 0 ] = new_x; }
    void set_y( double new_y ) { coord[ 1 ] = new_y; }
    void set_z( double new_z ) { coord[ 2 ] = new_z; }

    void set_coordinate( int i, double new_c )
    void set_coordinate( int i, double new_c )
    { coord[ i ] = new_c; }

    // Position operators
    // Get displacement, i.e. a vector, as difference of
    // two positions.
    friend DECL_KERN SPAvector operator-( SPAposition const &,
        SPAposition const & );

    // Translate a position by a vector.
    friend DECL_KERN SPAposition operator+( SPAposition const &,
        SPAvector const & );
    friend DECL_KERN SPAposition operator+( SPAvector const &,
        SPAposition const & );
    SPAposition const &operator+=( SPAvector const & );
    friend DECL_KERN SPAposition operator-( SPAposition const &,
        SPAvector const & );
    SPAposition const &operator-=( SPAvector const & );

...

    // Transform a position.
    friend DECL_KERN SPAposition operator*( SPAmatrix const &,
        SPAposition const & );
    friend DECL_KERN SPAposition operator*( SPAposition const &,
        SPAmatrix const & );
    SPAposition const &operator*=( SPAmatrix const & );

    friend DECL_KERN SPAposition operator*( SPAposition const &,
        SPAtransf const & );
    friend DECL_KERN SPAposition operator*( SPAposition const &p,
        SPAtransf const *t );
    SPAposition const &operator*=( SPAtransf const & );

...
```

```
    // Output details of a position
    void debug( FILE * = debug_file_ptr ) const;

};
```

Example 2-1. Declaration of SPAposition Class

Example 2-2 shows a code fragment of two attempts to use the SPAposition class. As the declaration of the SPAposition class in Example 2-1 shows, this has private data which is an array of three real numbers, called coord. Attempting to access the private data member coord directly fails at compile time. However, accessing the public methods, such as x, is valid and will compile.

C++ Example

```
// Declare an instance of the class.
SPAposition my_p1(10, 15, 20);

// The compile returns an error in next statement because
// ”coord” is the private data member of this class.
double my_x = my_p1.coord[0];

// The correct way to access the x coordinate of the position.
double my_x = my_p1.x();
```

Example 2-2. Accessing SPAposition Class Data

Overloading Class Constructors

Topic: *C++ Interface

Overloading class constructors means that there is more than one way to create a class instance. In ACIS, additional constructors are made available as a convenience based on the type of data that an application programmer may have on hand at the time a class instance is to be created.

The decision of which constructor to use happens at compile time, based on the argument types passed into the constructor.

Consider again the SPAposition class declared in Example 2-1. A SPAposition instance can be created from three real numbers, from an array of three real numbers, or from another SPAposition instance. This is illustrated in Example 2-3.

C++ Example

```
// Create an instance of SPAposition using 3 real numbers.
SPAposition p1 (3, 4, 5);

// Create an instance of SPAposition using an array.
double my_array[3] = { 3, 4, 5};
SPAposition p2 (my_array);

// Create an instance of SPAposition using a previously defined
// instance of SPAposition.
SPAposition p3 (p1);
```

Example 2-3. Overloaded Constructors

Copying Objects

Topic: *C++ Interface

ACIS provides class copy constructors, class methods, and functions for copying objects. A copy of an object may be a deep copy or a shallow copy.

A deep copy is a copy made by replicating the object plus any assets owned by the object, including objects pointed at by data members of the object being copied. The copied item does not share any data with the original. A deep copy allocates new storage for all member data and any pointers, so that all the information about that item is self–contained in its own memory block.

A shallow copy copies an object, but instead of copying all the other objects it references (a deep copy) it references the same objects that the original uses. A shallow copy stores only the first instance of the item in memory, and increments a reference count for each copy.

For a copy of an entity that does not share underlying information with the original, a deep copy should be used. One reason for using a deep copy of an entity is to move the deep copy into a different history stream, breaking all ties with the previous history stream. A call to api_deep_copy_entity is used to create the deep copy. There are some entities that are can not be deep copied (by design). If such entities are present during a deep copy, a sys_error will be thrown. A flag can be passed into the API if attributes that can't be deep copied need to be skipped over when doing a deep copy. Any non–attribute entities that cannot be deep copied will throw a sys_error regardless of the logical flag setting.

Overloading Class Methods and Operators

Topic: *C++ Interface

ACIS overloads operators, such as the minus operator (–). This permits the proper handling of elements, depending on the particular class. For example, when one position is subtracted from another, the result is the vector that goes between the two positions, as is shown by the instance of the SPAvector class in Example 2-4.

C++ Example

```
SPAposition p1 (3, 4, 5);
SPAposition p2 (6, 7, 8);
SPAvector v1 = p2 – p1;
```

Example 2-4. Overloaded Operators

ACIS makes use of the fact that C++ is a strongly typed to enforce rules of combination. For example, the concept of adding two positions does not make sense. However, it does make sense to create a new position that is the sum of a position and a vector, as is shown in Example 2-5, which creates a new instance of the SPAposition class by adding a previous SPAposition instance to a SPAvector instance.

C++ Example

```
SPAposition p1 (3, 4, 5);
SPAposition p2 (6, 7, 8);
SPAposition p3 = p1 + p2;  // Error: this does not compile
SPAvector v1 (10, 20 , 30);
SPAposition p4 = p1 + v1;  // creates a new position offset from
                           // position p1 by vector v1.
```

Example 2-5. Using C++ Strong Types

Overloading Class new and delete

Topic: *C++ Interface

ACIS overloads the C++ new operator to allocate space on the portion of the heap controlled by ACIS. This is used in conjunction with the other constructors. Each class derived from ENTITY defines its own new and delete operators that the ACIS free list manager. (These operators are supplied by the ENTITY_FUNCTIONS and UTILITY_DEF macros.)

When working with Microsoft Developer Studio and/or MFC, the beginning of the file may have been enhanced by Microsoft during a _DEBUG compile. Sometimes you will see:

```
#ifdef _DEBUG
#define new DEBUG_NEW
// ... other code...
#endif
```

This interferes with the ACIS overloading of new for the entities. So you have two options. Either remove the offending define or move the code that creates your ENTITY objects to another file.

In fact it is an issue when trying to instantiate any object derived from an ACIS entity in MFC. ACIS overrides new and delete for all entity objects and thus one cannot use the standard new operator. Using the new operator for ENTITY derivations needs to have API_BEGIN/API_END around the ENTITY creation code, so it is best to do the allocation in an API function.

```
outcome api_make_attrib(ENTITY*& ent,
    ATTRIB_GEN_INTEGER*& attrib,
    char*& name, int& value)
{
    API_BEGIN
#undef new
    attrib = new ATTRIB_GEN_INTEGER(ent,
        name, value);
#define new DEBUG_NEW
    API_END
}
```

Overloading API Functions

Topic: *C++ Interface

Although the discussion so far in this chapter has been about C++ classes and their associated data elements and methods, much of this holds true for the API function interface.

The API functions form the primary interface to ACIS and add an additional level of interface stability. Class methods may change from release to release. However, the API interface remain consistent from release to release. The API functions are often wrappers for C++ class methods or groupings of methods.

Like C++ class methods, API functions can be overloaded. This means that a single name can refer to multiple forms of an API function, with the differences being in the number of arguments and/or in the argument types. The distinction between the overloaded forms of API functions happens at compile time.