NAME

set, bag, sequence, array – set bag, sequence, and array attributes

SYNOPSIS

```
// in sdl:
interface a; // some object class definition
interface my_obj {
public:
    attribute set<a> a_set;
    attribute bag<a> a_bag;

    // sdl permits only sequences of values
    attribute sequence<ref<a>> a_seq;

    // variable-length sequence:
    attribute sequence<long> vseq;
    attribute sequence<long, 3> bounded_vseq;

    attribute long an_array[3];
};

// The C++ language binding for my_obj (with
// the private portion omitted):

class           my_obj:public sdlObj {
public:
    Set < a > a_set;
    Bag < a > a_bag;
    sdl_val_set_apply < Ref < a > >a_seq;
    sdl_val_set < long >vseq;
    sdl_val_set < long >bounded_vseq;
    long            an_array[3];
};


// signature of C++ bindings for sets, bags, and sequences:
// (The public potions of their class declarations.)

template <class T>
class Set
{
public:
    void add(const Ref<T>);
    void del(const Ref<T> &arg); // delete the given elt
    Ref<T> delete_one(); // delete any elt and return it
    Ref<T> get_elt(int i) const;
    size_t get_size(); // size_t is defined in <sys/types.h>
    bool member(const Ref<T>  &arg) const;
};

template <class T>
class Bag
{
public:
```

```
    void add(const Ref<T>);
    void del(const Ref<T> &arg); // delete the given elt
    Ref<T> delete_one(); // delete any elt and return it
    Ref<T> get_elt(int i) const;
    size_t get_size(); // size_t is defined in <sys/types.h>
    bool member(const Ref<T>  &arg) const;
};

// for sequences:

template <class t>
class Sequence {
public:
    size_t max_elements() const { return cur_size / sizeof(t); }
    int append_elt(const t & new_elt); //initialized element
    int append_elt();    // uninitialized element
    void delete_elt(unsigned int i);
    const t & get_elt(unsigned int i) const;
    const t & operator[] (unsigned int i) const;
    t & write_elt(unsigned int i);
    size_t get_size() const; // returns number of elements
    void set_size(int i) ; // sets size absolutely
};
```

DESCRIPTION

Sets, bags, and sequences are types that can be used for *attributes* of objects (SDL interfaces). (The template classes that implement sets, bags, or sequences, are implementations for object attributes, and they cannot be used for *transient* data structures.)

The SDL set and bag types provide a mechanism for maintaining a collection of references to Shore objects. In the SDL source, the members, or elements of sets and bags are declared to be object types (types declared as SDL interfaces); the set or bag is implemented by reference. That is, the set attribute stores a set of object references and not copies of the objects. Attributes of SDL objects declared as type *set*<*T*> can be accessed within SDL/C++ programs as if they were instances of the C++ template class *Set*<*T*> with the method signature shown above. The only difference between *set*<*T*> and *bag*<*T*> is that an instance of *set*<*T*> never contains more than one reference to a particular object of type *T;* an attribute of type *bag*<*T*> may contain any number of references to the same object. That is, sets never contain duplicate references, while bags may contain duplicates.

The SDL sequence type can be used to provide a variable-length array abstraction within an SDL object. If an attribute is declared as sequence<T>, values can be appended to the end of the sequence and elements within the sequence can be accessed using a simple integer index as their location within the sequence. Unlike sets and bags, sequences store elements by value, not by references; therefore, the type parameter of a sequence type may not be an object (interface) type. The declaration of a sequence attribute may include an upper bound on the length of the sequence, which allows the language binding to use an alternative, possibly more efficient, implementation for sequences when a bound is known. (The language binding illustrated here does not do so.)

The SDL array type is a fixed-length array.

In what follows, we assume bindings for the following SDL definitions in the synopsis above. We also assume the following C++/SDL variable declarations:

```
Ref<a> a_ref;
Ref<my_obj> o_ref;
```

SET AND BAG OPERATIONS

The **add** method inserts a reference of the correct type into the given set:

```
o_ref.update()->a_set.add(a_ref);
```

adds the reference *a_ref* to the *a_set* attribute of the object to which the variable *o_ref* refers.

To remove an element, use **del:**

```
o_ref.update()->a_set.del(a_ref);
o_ref.update()->a_bag.del(a_ref);
```

deletes any reference to the element *a_ref* or the first value matching *a_ref* (for bags). If no matching reference is found within the set/bag, the operation has no effect.

To delete and return elements one at a time, use **delete_one:**

```
a_ref = o_ref.update()->a_set.delete_one();
```

deletes one element of a set, and returns a ref to the element deleted; the element chosen for deletion is based on efficiency considerations.

The **get_size** function returns the number of elements in the set or bag.

The method **member** returns *true* if the given element is a member of the set or bag, *false* otherwise.

Iterating over Sets and Bags

A simple idiom for iterating though the elements of a set or bag is is to increment a parameter to **get_elt** until a null reference is returned, as shown below.

```
int i;
for (a_ref = o_ref->a_bag.get_elt(i=0);
    a_ref != NULL;
    a_ref = o_ref->a_bag.get_elt(++i))
{
    // do something with a_ref
}
```

The **get_elt** function returns a reference to the *i*th element of the set, numbered from 0 to n-1, where n-1 is the value returned by **get_size;** if the parameter is out of range for the set, a null value is returned. If an application inserts or deletes members while iterating in this fashion, the results are not defined.

This interface is likely to change in a future release, to provide a more set-like mechanism for iteration.

SEQUENCE OPERATIONS

Sequence attributes have operations similar to sets and bags, but they store values, not references to objects. Object types (interfaces) may not be used as the element type in a sequence declaration.

Sequence attributes are initially empty. In this respect, even those declared with an upper bound differ from fixed-length arrays.

A sequence may be extended, one element at a time, using **append_elt.** Two overloaded variants are provided, both of which return the index of the new element. If the parameterless variant is called, the new element is uninitialized. If a parameter of the sequence element type is provided, the value of the parament is used to initialize the new sequence element.

```
  int i;
i = o_ref.update()->a_seq.append_elt();  // uninitialized
i = o_ref.update()->a_seq.append_elt(a_ref);

i = o_ref.update()->vseq.append_elt();  // uninitialized
i = o_ref.update()->vseq.append_elt(2);
```

Random access to the elements of a sequence is made with an integer index (with origin 0). The function **get_elt** returns a const C++ reference to the *i*th element of the sequence, that is, it allows read only access to the element. The member function **operator[]** is semantically identical to **get_elt.**

The method **write_elt** returns a non-const C++ reference to the *i*th element; this form may be used to update an element of a sequence in place. **Get_size** returns the number of elements in the sequence, and **set_size** changes the number of elements in the sequence. If set_size causes a sequence to increase in size, the new elements are not initialized.

Elements may be deleted from a sequence using **delete_elt,** which removes the *i*th element from the sequence:

```
long i;
if(o_ref->a_seq.get_size() == 2) {
    // these 2 are identical :
    a_ref = o_ref->a_seq.get_elt(0);
    a_ref = o_ref->a_seq[0];

    o_ref.update()->a_seq.delete_elt(0); // renumbers element [1] to [0]
    o_ref.update()->a_seq.delete_elt(0); // removes what was element [1]
}

// Similarly:
if(o_ref->vseq.get_size() == 2) {
    // these 2 are identical :
    i = o_ref->vseq.get_elt(0);
    i = o_ref->vseq[0];

    o_ref.update()->vseq.delete_elt(0); // renumbers element [1] to [0]
    o_ref.update()->vseq.delete_elt(0); // removes what was element [1]
}
```

This results in the renumbering of the elements in the sequence that follow the one deleted. If any of the functions **write_elt, operator[],** or **get_elt** tries to access an element beyond the range of the sequence, the first element in the sequence is returned, since there is no way to return a "null" value. It is up to the application program to check the validity of indexes prior to using them.

ARRAY OPERATIONS

The language binding for an array attribute is straightforward: it is an array of the type seen in the SDL declaration. Access to the values is as expected for any array:

```
long i;
o_ref.update()->an_array[0] = 1;
i = o_ref->an_array[0];
```

The application must explicitly initialize the elements of an array.

RESTRICTIONS

There isn't any way to define operator==() in SDL; thus set elements are restricted to primitive types (e.g. long, float, etc.) or references to objects (interface) types.

NOTE

The implementation of sets, bags, and sequences is somewhat simplistic; this implementation will likely be superseded by an implementation based on the C++ Standard Template Library.

VERSION

This manual page applies to Version 1.1 of the Shore software.

SPONSORSHIP

The Shore project is sponsored by the Advanced Research Project Agency, ARPA order number 018 (formerly 8230), monitored by the U.S. Army Research Laboratory under contract DAAB07-91-C-Q518.

COPYRIGHT

SEE ALSO

intro(cxxlb), intro(oc), method(cxxlb), index(cxxlb), ref(cxxlb), and the **Shore Data Language Reference Manual**