ObjectStore® PSE Pro™

PSE Pro for C++ Collections Guide and Reference

Release 6.3

PROGRESS SOFTWARE | Real Time Division

PSE Pro for C++ Collections Guide and Reference, Release 6.3, October 2005

September 2005

Contents

Preface

Purpose The *PSE Pro for C++ Collections Guide and Reference* describes how to use the C++ programming interface to PSE Pro for C++ to allocate, populate, and manipulate collections.

Audience This book assumes you are experienced with C++.

Scope Information in this book assumes that PSE Pro is installed and configured.

How This Book Is Organized

Chapter 1, Introducing Collections, on page 15, introduces collections, describes the different types of collections, and provides a simple example.

Chapter 2, Performing Basic Collections Functions, on page 27, discusses how to create, destroy, populate, and obtain information about collections.

Chapter 3, Using Cursors to Navigate Collections, on page 35, describes how to use a cursor to iterate over the elements in a collection.

Chapter 4, Using Dictionaries, on page 41, provides information about collections that use keys to keep track of elements.

Chapter 5, Performing Advanced Collections Operations, on page 53, discusses the use of paths to navigate to collection elements, the techniques for controlling collection traversal, supplying rank and hash functions, and specifying expected collection size.

Chapter 6, Class Reference, on page 61, provides complete information about each class in the collections facility. Information is presented in alphabetical order by class name.

Chapter 7, Macros and User-Defined Functions Reference, on page 149, presents information in alphabetical order about each collections macro and user-defined function.

Notation Conventions

This document uses the following conventions:

Convention	*Meaning*
`Courier`	`Courier font` indicates code, syntax, file names, API names, system output, and the like.
`Bold Courier`	**`Bold Courier font`** is used to emphasize particular code, such as user input.
`Italic Courier`	*`Italic Courier font`* indicates the name of an argument or variable for which you must supply a value.
Sans serif	Sans serif typeface indicates the names of user interface elements such as dialog boxes, buttons, and fields.
Italic serif	In text, *italic serif typeface* indicates the first use of an important term.
`[ ]`	Brackets enclose optional arguments.
`{ }*` or `{ }+`	When braces are followed by an asterisk (*), the items enclosed by the braces can be repeated `0` or more times; if followed by a plus sign (+), one or more times.
`{` *`a`* `|` *`b`* `|` *`c`* `}`	Braces enclose two or more items. You can specify only one of the enclosed items. Vertical bars represent OR separators. For example, you can specify *a* or *b* or *c*.
`...`	Three consecutive periods can indicate either that material not relevant to the example has been omitted or that the previous item can be repeated.

Progress Software Real Time Division on the World Wide Web

The Progress Software Real Time Division Web site (`www.progress.com/realtime`) provides a variety of useful information about products, news and events, special programs, support, and training opportunities.

Technical Support

To obtain information about purchasing technical support, contact your local sales office listed at `www.progress.com/realtime/techsupport/contact`, or in North America call 1-781-280-4833. When you purchase technical support, the following services are available to you:

- You can send questions to `realtime-support@progress.com`. Remember to include your serial number in the subject of the electronic mail message.
- You can call the Technical Support organization to get help resolving problems. If you are in North America, call 1-781-280-4005. If you are outside North America, refer to the Technical Support Web site at `www.progress.com/realtime/techsupport/contact`.
- You can file a report or question with Technical Support by going to `www.progress.com/realtime/techsupport/techsupport_direct`.
- You can access the Technical Support Web site, which includes
  - A template for submitting a support request. This helps you provide the necessary details, which speeds response time.

- Solution Knowledge Base that you can browse and query.
- Online documentation for all products.
- White papers and short articles about using Real Time Division products.
- Sample code and examples.
- The latest versions of products, service packs, and publicly available patches that you can download.
- Access to a support matrix that lists platform configurations supported by this release.
- Support policies.
- Local phone numbers and hours when support personnel can be reached.

Education Services

To learn about standard course offerings and custom workshops, use the Real Time Division education services site (`www.progress.com/realtime/services`).

If you are in North America, you can call 1-800-477-6473 x4452 to register for classes. If you are outside North America, refer to the Technical Support Web site. For information on current course offerings or pricing, send e-mail to `classes@progress.com`.

Searchable Documents

In addition to the online documentation that is included with your software distribution, the full set of product documentation is available on the Technical Support Web site at `www.progress.com/realtime/techsupport/documentation`. The site provides documentation for the most recent release and the previous supported release. Service Pack README files are also included to provide historical context for specific issues. Be sure to check this site for new information or documentation clarifications posted between releases.

Your Comments

Real Time Division product development welcomes your comments about its documentation. Send any product feedback to `realtime-support@progress.com`. To expedite your documentation feedback, begin the subject with `Doc:`. For example:

```
Subject: Doc: Incorrect message on page 76 of reference manual
```

Chapter 1
Introducing Collections

A collection is an object whose purpose is to group together other objects. It provides a convenient means of storing and manipulating groups of objects by supporting operations for inserting, removing, and retrieving elements.

This chapter discusses the following topics:

Overview of Collections

The ObjectStore collections facility provides

- A library of collection classes
- Facilities that allow traversal, manipulation, and retrieval of the elements within collections

The collections facility is rich and varied. Consequently, it adds overhead to your code and database. If your application does not require collections, a simple linked list you write yourself might be a more suitable choice.

Collections Class Library

The classes in the ObjectStore collections class library provide the data structures for representing several types of collections. The categories of classes and their different behaviors are as follows:

- Sets do not maintain order, do not allow nulls, and ignore duplicates.
- Bags do not maintain order and do not allow nulls, but they do allow duplicates.
- Lists maintain order and do not allow nulls, but they do allow duplicates.
- Arrays maintain order and allow nulls and duplicates.
- Dictionaries associate a key with each element or group of elements. They can be ordered or unordered, do not allow nulls, and ignore duplicates.
- Index-only collections support `O(1)` element look-up. The index can be ordered or unordered, supports null insertions, and ignores duplicate insertions.

An object can be contained in many different collections, and collections can be used in transient or persistent memory, depending on the needs of your application.

Collections Manipulation Features

Each class defines member functions that allow you to insert, remove, and count the elements in the collection. With the ObjectStore cursor class, you can create a cursor to iterate over the elements in a collection and to retrieve elements for examination or processing one at a time.

Collections are commonly used to model many-valued attributes, and they can also be used as class extents (which hold all instances of a particular class). Collections of one type — *dictionaries* — associate a key with each element or group of elements, and so can be used to model binary associations or mappings. ObjectStore dictionaries are described in detail in Chapter 4, Using Dictionaries, on page 41.

Requirements for Collections Applications

The following sections describe the requirements for applications that use collections:

- Including Collections Header Files
- Initializing the Collections Facility
- Generating the Application Schema
- Linking with ObjectStore Libraries

Including Collections Header Files

After including the standard ObjectStore header file <os_pse/ostore.hh>, programs that use ObjectStore collections must include the header file <os_pse/coll.hh>.

If your application uses ObjectStore dictionaries, your program must include <os_pse/coll/dict_pt.hh> and must also include <os_pse/coll/dict_pt.cc> in any source file that instantiates an `os_Dictionary`, following the other header files.

Initializing the Collections Facility

To use the collections facility, after an application calls the `objectstore::initialize()` function, it must call the static member function `os_collection::initialize()`. For example:

```
objectstore::initialize();
os_collection::initialize();
```

Generating the Application Schema

As with any ObjectStore application, applications that use the collections facility must generate an application schema. Create a schema source file that marks the classes you want to store in the database.

If you use parameterized persistent collections, you must mark those collection types in your schema source file.

If you use persistent dictionaries, you must call the macro `OS_MARK_DICTIONARY()` in the schema source file for each key-type/element-type pair that you use. Calls to this macro have the form

`OS_MARK_DICTIONARY(`*`key-type`*`, `*`element-type`*`)`

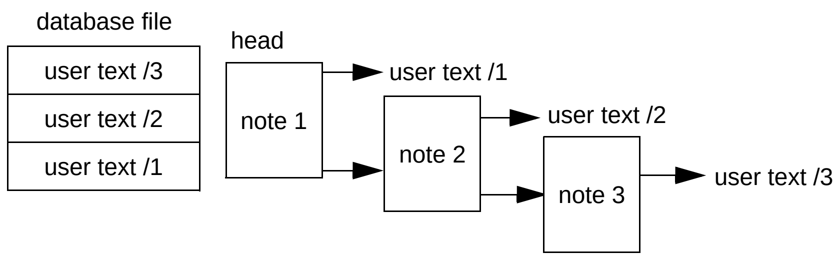
Specific information about marking dictionaries can be found in Marking Persistent Dictionaries on page 43. See also `OS_MARK_DICTIONARY()` on page 150.

Linking with ObjectStore Libraries

Programs that use ObjectStore collections must link with the `ostore.lib` library.

Introductory Collections Example

Here is a simple example to illustrate how to use collections. It defines a class that records a note entered by the user. Notes are maintained in an `os_list` in reverse order from that in which they were created; that is, the most recent note is at the beginning of the list. At start-up, the database file is read and the notes are created in memory. The existing notes are displayed and the user is prompted to enter a new note. The database file is rewritten starting with the new note.



Header file: note.hh

```
#include <ostore/ostore.hh>

class note {
  public:

    /*  Public Member functions */
    note(const char*, note*, int);
    ~note();
    void display(ostream& = cout);
    static os_typespec* get_os_typespec();

    /* Public Data members */
    char* user_text;
    note* next;
    int   priority;
};
```

To establish an entry point, an `os_database_root` called `root_head` is assigned and points to the value returned from the `find_root` member function defined on the class `os_database`. The `head` variable is assigned to point to the value returned by the `get_value` function applied to `root_head`.

Each note instance and its user text is allocated persistently in an `os_list`, and a transaction surrounds the code that touches persistent data. The value of the database root is set to the `head` of the linked list.

Header file: note.hh

```
#include <iostream>
#include <string.h>
#include <os_pse/ostore.hh>
#include <os_pse/coll.hh>

/* A simple class that records a note entered by the user. */

class note {

  public:

    /* Public Member functions */
    note(const char*, int);
    ~note();
    void display(ostream& = cout);
```

```
    static os_typespec* get_os_typespec();

    /* Public Data members */
    char* user_text;
    int  priority;
};
```

Main program: main.cc

```
/* ++ Note Program - main file */

#include "note.hh"
extern "C" void exit(int);
extern "C" int atoi(char*);

/* Head of linked-list of notes */
os_list *notes = 0;
const int note_text_size = 100;

main(int argc, char** argv) {

  if(argc!=2) {
    cout << "Usage: note <database>" << endl;
    exit(1);
  } /* end if */

  objectstore::initialize();
  os_collection::initialize();
  char buff[note_text_size];
  char buff2[note_text_size];
  int  note_priority;

  os_database *db = os_database::open(argv[1], 0, 0644);

  OS_BEGIN_TXN(t1,0,os_transaction::update) {

    os_database_root *root_head = db->find_root("notes");
    if(!root_head)
      root_head = db->create_root("notes");
    notes = (os_list *)root_head->get_value();

    if(!notes) {
      notes = new(db, os_list::get_os_typespec()) os_list;
      root_head->set_value(notes);
    } /* end if */

    os_cursor c(*notes);
    /* Display existing notes */
    for(note* n=(note *)c.first(); n; n=(note *)c.next())
      n->display();

    /* Prompt user for a new note */
    cout << "Enter a new note: "<< flush;
    cin.getline(buff, sizeof(buff));

    /* Prompt user for a note priority */
    cout << "Enter a note priority: "<< flush;
    cin.getline(buff2, sizeof(buff2));
    note_priority = atoi(buff2);

    notes->insert(new(db, note::get_os_typespec())
      note(buff, note_priority) );
  }
  OS_END_TXN(t1)

  db->close();
}
```

Choosing a Collection Type

This section contains a brief description of each type of ObjectStore collection, followed by a simple decision tree you can use to choose a collection type to suit your program's particular behavioral requirements.

All collection types described below (with the exception of `os_Dictionary`) have both a templated (parameterized) and a nontemplated (nonparameterized) version. For ease of differentiation, the templated versions use uppercase letters (for example, `os_Set`), whereas the nontemplated versions use lowercase (`os_set`). Nontemplated classes are always typed as `void*` pointers.

- For information about the differences between templated (parameterized) and nontemplated (nonparameterized) collection classes, see Templated and Nontemplated Collections.
- For information about the characteristics of ObjectStore collection classes, see Collection Characteristics and Behaviors.
- For a description of hierarchical representation of the relationships between the ObjectStore collection types, see the Class Hierarchy Diagram.
- For information about how to create collection classes, see Creating Collections on page 28.

os_Set and os_set

Sets, as with familiar data structures such as linked lists and arrays, have *elements.* Elements are objects that the set groups together. But, in contrast to the elements of lists and arrays, the elements of a set are unordered. Use sets to group objects together when you do not need to record any particular order for the objects.

Besides lacking order, sets do not allow multiple occurrences of the same element. This means that inserting a value that is already an element of a set leaves the set unchanged.

os_Bag and os_bag

Bags are collections that keep track of what their elements are and also of the number of occurrences of each element. In other words, bags allow duplicate elements. Bags provide all the operations available for sets, as well as an operation, `count()`, that returns the number of occurrences of a given element in a given collection.

os_List and os_list

In addition to sets and bags, the ObjectStore collections facility supports *lists*. A list is a collection that associates a numerical position with each element based on insertion order. Lists allow duplicates. In addition to simple insert (insert into the beginning or end of the collection) and simple remove (removal of the first occurrence of a specified element), you can insert, remove, and retrieve elements based on a specified numerical position, or based on a specified cursor position (see Accessing Collection Elements with a Cursor or Ordinal Value on page 39).

`os_nList` and `os_nlist` are derived from parameterized `os_List` and `os_list`, respectively. They allow the user to customize the internal representation of ObjectStore lists.

os_Array and os_array

ObjectStore *arrays* are like ObjectStore lists, except that they always provide access to collection elements in constant time. That is, the time complexity of operations such as retrieval of the n^{th} element is order `1` in the array's size. Arrays also always allow null elements, and they provide the ability to automatically establish a specified number of new null elements.

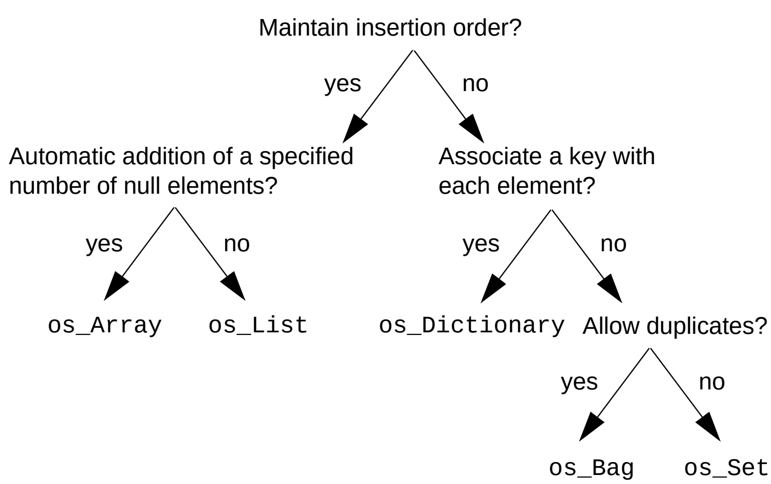
os_Dictionary

Like bags, ObjectStore *dictionaries* are unordered collections that allow duplicates. Unlike bags, however, dictionaries associate a *key* with each element. The key can be a value of any C++ fundamental type, a user-defined type, or a pointer type. When you insert an element into a dictionary, you specify the key along with the element. You can retrieve an element with a given key, or retrieve those elements whose keys fall within a given range.

Dictionaries are somewhat different from other ObjectStore collection classes in their use of keys. See Chapter 4, Using Dictionaries, on page 41, for additional information on how dictionaries differ from other kinds of ObjectStore collections.

Using a Decision Tree to Select a Collection Type

Here is a simple decision tree to help you choose a collection type to suit particular behavioral requirements.



Collection Characteristics and Behaviors

To use collections, it is important to understand the characteristics and behavior of each type of collection. The topics discussed in this section are

- Collections Store Pointers to Objects
- Collections Can Be Transient or Persistent
- Parameterized and Nonparameterized Collections
- Class Hierarchy Diagram
- Collection Behaviors
- Expected Collection Size
- Performing pick() on an Empty Collection

Collections Store Pointers to Objects

ObjectStore collection classes store pointers to objects, not the objects themselves. Thus, elements exist independently from membership in a collection. A single element can be a member of many collections.

Collections Can Be Transient or Persistent

Like all types in ObjectStore, collections can be used in transient memory (program memory) or persistent memory. Transient collections are used to represent transient, changeable groupings; the current list of cars in the parking garage, for example. Persistent collections contain more permanent associations, such as the list of people on a board of directors or the founding states of the European community.

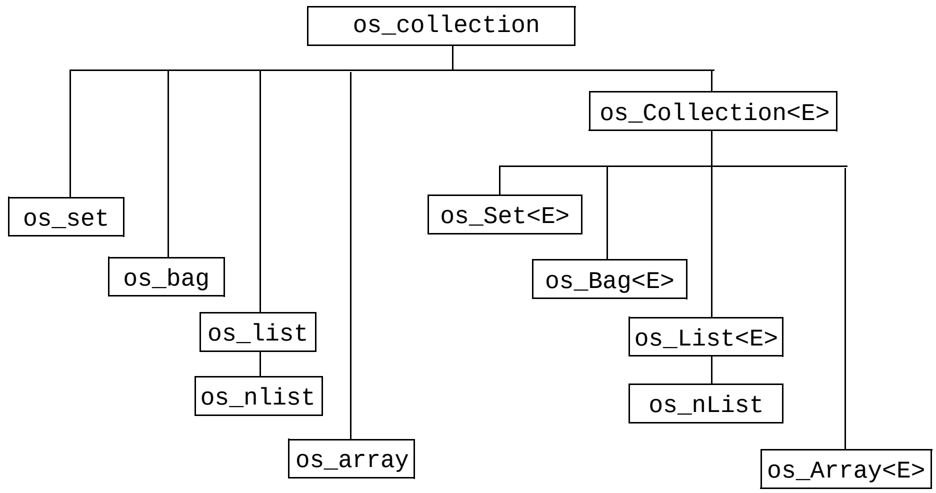
Parameterized and Nonparameterized Collections

Every ObjectStore collection class (except `os_Dictionary`) is provided in both a templated (parameterized) and a nontemplated (nonparameterized) form. See Templated and Nontemplated Collections on page 24.

Templated classes use an uppercase letter in their class names (`os_Set`), whereas nontemplated classes use lowercase letters in their class names (`os_set`).

Class Hierarchy Diagram

The following diagram shows the hierarchical relationship among most (`os_Dictionary`, `os_nlist`, and `os_nList` are not included) of the public ObjectStore collection classes. Note that `E` is actually a pointer value: the *element type parameter* used by the templated collection classes to specify the types of values allowable as elements. See Using Collections with the Element Type Parameter on page 24 for more information.



Collection Behaviors

The ObjectStore collection classes vary according to what behaviors and characteristics are permitted and prohibited. The following table lists the classes and their default behaviors. You can change the default size of a collection; see Chapter 5, Performing Advanced Collections Operations, on page 53, for more information on customizing your ObjectStore collections

Collection Class	*Maintain Element Order*	*Allow Duplicates*	*Allow Nulls*
`os_Set`	No	No	No
`os_Bag`	No	Yes	No
`os_List`	Yes	Yes	No
`os_Array`	Yes	Yes	Yes

The `os_Dictionary` class differs substantially from other collection classes in its behaviors. Dictionary behaviors are related to the *key* of an element rather than to an element itself. See Chapter 4, Using Dictionaries, on page 41, for information on how ObjectStore dictionaries differ from other ObjectStore collection classes.

The behavior of `os_nList` and `os_nlist` is the same as that of `os_List`. See `os_nList and os_nlist` on page 137.

Expected Collection Size

By default, all collection classes are presized with a representation suitable for a size of less than 20. The `expected_size` argument for the collection constructors lets you supply a different default size. For more information, see Specifying Expected Size on page 60.

Performing pick() on an Empty Collection

For all collection classes, performing `pick()` on an empty collection or on an empty result of a query results in a null return value.

Templated and Nontemplated Collections

ObjectStore collection classes are provided in both templated (parameterized) and nontemplated (nonparameterized) versions.

Using Collections with the Element Type Parameter

The parameterized ObjectStore collection classes — `os_Set`, `os_Bag`, `os_List`, `os_Collection`, `os_Dictionary`, and `os_Array` — are class templates. Each class has a parameter, the *element type parameter*, that specifies the type of value allowable as elements. This type must be a pointer type. For example:

```
os_Set<part*> &a_part_set =
  *new(db1, os_Set<part*>::get_os_typespec()) os_Set<part*>;
```

This fragment defines a variable whose value is a reference to a set of pointers to `part` objects. The element type, `part*`, appears in angle brackets when the collection type is mentioned. (Note that the element type parameter is represented as `<E>` in function signatures.)

`os_nList` and `os_nlist` are also parameterized ObjectStore collections classes. The parameters for these classes are used to customize the internal representation of the list. See `os_nList and os_nlist` on page 137.

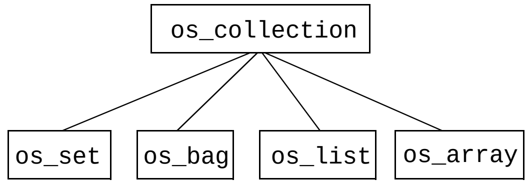
Example: os_Set

The following example uses an instance of `os_Set`, one of the classes supplied by the collections facility. This class defines a part that includes the part number and the responsible engineer.

```
#include <os_pse/ostore.hh>
#include <os_pse/coll.hh>

class part {

  public:

    int part_number;
    os_Set<part*> &children;
    employee *responsible_engineer;

    part (int n) :
      children(*new(
            db1,
            os_Set<part*>::get_os_typespec())
          os_Set<part*> {
        part_number = n;
        responsible_engineer = 0;
      }

};
```

Using Collections Without Parameterization

You can choose to use the following nonparameterized collection classes:



Notice that the names of the parameterized classes have an uppercase letter following the `os_`, while the nonparameterized classes have a lowercase letter following the `os_`. Notice, as well, that there is no nonparameterized version of `os_Dictionary`.

Difference Between Parameterized and Nonparame-terized Interfaces

The difference between the parameterized and nonparameterized interfaces is that with the parameterized interface, you can inform the compiler of the type of element a collection is to have, allowing the compiler to provide an extra measure of type safety. With the nonparameterized interface, elements are always typed as `void*` pointers. Most of the examples in this manual use the parameterized interface, but if you are not using parameterization, just drop any construct of the form

<*element-type-name*>

and use the nonparameterized collection type names (beginning with `os_` followed by a lowercase letter). If you use `os_set` instead of `os_Set`, class definition example used earlier in Using Collections with the Element Type Parameter on page 24 would then look like the following:

Example: os_set()

```
class employee ;
extern os_database *db1 ;
class part {
  public:
    int part_number ;
    os_set &children ;
    employee *responsible_engineer ;
    part (int n) :
      children(*new(db1, os_set::get_os_typespec()) os_set {
        part_number = n;
        responsible_engineer = 0 ;
    }
};
```

In this example, `os_Set`, and `os_Set` is changed to `os_set` and the element type parameter `<part*>` is left out.

Nonparameterized collections are typed void*

When you use nonparameterized collections, elements are typed `void*`. This means you must cast the result of retrieving a collection element, for example, when using `pick()`or traversing a collection by using a cursor.

Chapter 2 Performing Basic Collections Functions

A collection is an object whose purpose is to group together other objects. It provides a convenient means of storing and manipulating groups of objects by supporting operations for inserting, removing, and retrieving elements.

This chapter discusses the following topics:

Creating Collections

This section presents information on creating collections. The `os_Array`, `os_Bag`, `os_Collection`, `os_List`, and `os_Set` classes are discussed together in the first subsection. `os_Dictionary` is discussed separately in Chapter 4, Using Dictionaries, on page 41.

General Guidelines

You create collections for each collection class with the following constructor functions:

Collection Type	*Constructor Function*
Array	`os_Array::os_Array()`
Bag	`os_Bag::os_Bag()`
List	`os_List::os_List()`
Set	`os_Set::os_Set()`

Constructor functions

The overloadings of constructor functions are listed in Chapter 6, Class Reference, on page 61. For each constructor, there is at least one overloading that creates an empty set and another that lets you specify the expected collection size. There are also two copy constructors that create a collection with the same membership as another specified collection.

Deleting Collections

To delete a collection, use the C++ `delete()` function. For example:

```
os_list *list = new(db, os_list::get_os_typespec()) os_list;
delete list;
```

You must explicitly delete each collection that you allocate. If you do not, your application has memory leaks.

Inserting Collection Elements

You update collections by inserting and removing elements or by using the assignment operators (see Copying, Combining, and Comparing Collections on page 33). The insert operations for `os_Collection` and its subtypes are declared as follows:

```
void insert(const E) ;
```

Inserting Dictionary Elements

For dictionaries, you specify an *entry*, that is, a key, along with the element to be inserted. Declare `os_Dictionary::insert()` as follows:

```
void insert(const K &key, const E element) ;
```

```
void insert(const K *key_ptr, const E element) ;
```

These two overloadings are provided for convenience so that you can pass either the key or a pointer to the key.

Caution For dictionaries with `signal_dup_keys` behavior, if an attempt is made to insert something with a key that already exists, `err_am_dup_key` is signaled.

Duplicate Insertions

For collections that do not support duplicates, inserting something that is already an element of the collection leaves the collection unchanged. For example:

```
os_database *db1 ;
message *a_msg ;
os_Set<message*> &temp_set =
    *new(db1, os_Set<message*>::get_os_typespec())
    os_Set<message*>;
. . .
temp_set.insert(a_msg) ;
temp_set.insert(a_msg) ;   /* set is unchanged */
```

For collections that support duplicates, each insertion increases the collection's size by 1 and increases by 1 the count (or number of occurrences) of the inserted element in the collection. You can retrieve the count of a given element with `count()`. Iteration with an unrestricted cursor visits each occurrence of each element.

Null Insertions

If you insert a null pointer (`0`) into a collection that does not support nulls, the collection remains unchanged and `err_coll_nulls` is signaled.

Ordered Collections

Inserting an element into an ordered collection adds the element to the end of the collection. See also Accessing Collection Elements with a Cursor or Ordinal Value on page 39.

Duplicate Keys

For dictionaries with `signal_dup_keys` behavior, if an attempt is made to insert an element with a key that already exists, `err_am_dup_key` is signaled.

Removing Collection Elements with remove()

The remove operations for `os_Collection` and its subtypes are declared in the following way:

```
os_int32 remove(E) ;
```

If you remove an element from a collection, the cardinality decreases by 1, and the count of the element in the collection decreases by 1. If you remove something that is not an element, the collection is unchanged. For example:

```
os_database *db1 ;
message *a_msg; . . .
os_Set<message*> &temp_set =
  *new(db1, os_Set<message*>::get_os_typespec())
    os_Set<message*>;
. . .
temp_set.remove(a_msg) ;
temp_set.remove(a_msg) ;   /* set is unchanged */
```

Ordered Collections

For collections that maintain order, `remove()` removes the specified element from the end of the collection. There are also overloadings of `remove()` that allow you to remove at a numerical index value in the collection or remove at the position of the cursor.

Removing Dictionary Elements

For dictionaries, you can also remove the entry with a specified key and element with `remove_value()`. This function is faster than `remove()`, so if you can specify the key, use `remove_value()` instead of `remove()`. There are two overloadings that differ only in that you pass a pointer to the key in one overloading and you pass a reference to the key in the other overloading.

```
void remove(const K &key, const E element) ;
```

```
void remove(const K *key_ptr, const E element) ;
```

If there is no entry with the specified key and element, the collection is unchanged. As with `remove()`, if you remove an entry from a dictionary, the cardinality decreases by 1, and the count of the element in the collection decreases by 1.

With dictionaries, you can also remove a specified number of entries with a specified key, using the following functions:

```
E remove_value(const K &key, os_unsigned_int32 n = 1) ;
```

```
E remove_value(const K *key_ptr, os_unsigned_int32 n = 1) ;
```

If there are fewer than `n` entries with the specified key, all entries in the dictionary with that key are removed. If there is no such entry, the dictionary remains unchanged.

Testing Collection Membership with contains()

To see if a given pointer is an element of a collection, use

```
os_int32 contains(E) const ;
```

This function returns nonzero if the specified `E` is an element of the specified collection, and `0` otherwise.

For dictionaries, you can determine if there is an entry with a given key and element.

```
os_boolean contains(
  const K const &key_ref,
  const E element
) const ;
```

```
os_boolean contains(
  const K *key_ptr,
  const E element
) const ;
```

With the first function, you pass a reference to the key; with the second, you pass a pointer to the key. Other than that, these two functions are equivalent. If there is no entry with the specified key and element, `0` (false) is returned.

Finding the Count of an Element with count()

To find the count of a given element in a collection that allows duplicates, use

```
os_int32 count(E) ;
```

If the specified `E` is not an element of the collection, `0` is returned.

For dictionaries, you can determine the number of entries with a specified key with one of these functions:

```
os_unsigned_int32 count_values(const K const &key_ref) const ;
```

```
os_unsigned_int32 count_values(const K *key_ptr) const ;
```

With the first function, you pass a reference to the key; with the second, you pass a pointer to the key. Other than that, these two functions are equivalent.

Finding the Size of a Collection with cardinality()

You can determine the number of elements in a collection with the member function `os_collection::cardinality()`.

```
os_unsigned_int32 cardinality() const ;
```

The cardinality of a collection that does not allow duplicates is the number of elements it contains. The cardinality of a collection that does allow duplicates is the sum of the number of occurrences of each of its elements.

You can test to see if a collection is empty with the member function `empty()`.

```
os_int32 empty() ;
```

This function returns true (a nonzero 32-bit integer) if it is empty, and false (`0`) otherwise.

Copying, Combining, and Comparing Collections

The class `os_Collection` defines several operators for assignment and comparison. Some of the assignment operators are related to the familiar set-theory operators union, intersection, and difference. In addition, some of the comparison operators are analogous to set-theory comparisons such as subset and superset. The collection operators are listed below. (LHS, below, stands for the operand on the left-hand side, and RHS stands for the right-hand-side operand.)

Collection operators

- `operator =()` replaces the contents of LHS with the contents of RHS.
- `operator |=()` adds the contents of RHS to LHS.
- `operator -=()` removes the contents of RHS from LHS.
- `operator &=()` replaces the contents of LHS with the intersection of LHS and RHS.
- `operator <()` (like proper subset).
- `operator >()` (like proper superset).
- `operator <=()` (like subset).
- `operator >=()` (like superset).
- `operator ==()` (checks if elements are the same).
- `operator !=()` (checks if any elements are different).

Dual Purpose of the Operators

All these operators have a dual purpose. They can be used on two collections, or they can be used on a collection (as left-hand operand) and an instance of that collection's element type (as right-hand operand). For example, they can be used on a set of parts and a part. In that case, the instance of the collection's element type is treated as a collection whose one and only element is that instance. Performance varies by representation. For example, you can use the union equals operator (`|=`) as a convenient way of performing inserts:

```
os_Set<message*> &a_set
  = *new(db1, os_Set<message*>::get_os_typespec())
      os_Set<message*>;
message *a_msg ;
. . .
a_set |= a_msg ;
```

And you can use `-=` to remove elements: `a_set -= a_msg ;`

Ordered Collections and Collections with Duplicates

When you use update operators, such as `|=`, on ordered collections or collections that allow duplicates, the result can be understood in terms of performing an iteration on one or more of the operands. So, for example:

```
big_list |= little_list ;
```

is equivalent to iterating through `little_list` in the default order, performing an insert into `big_list` for each occurrence of each element of `little_list`. Assignment of one collection to another as in

```
the_copy = the_original;
```

is equivalent to first removing all `the_copy`'s elements and then iterating through `the_original` in default order, performing an insert into `the_copy` for each occurrence of each element of `the_original`.

In general, the update operators (`=`, `|=`, `-=`, `&=`) bundle together a sequence of inserts or removes of elements of one or more operands *in the order in which those elements appear in the operands,* the default iteration order for the operands. This describes only the *behavior* of the operators. The implementations might be different. For example, to add all of a part's children to a given list, you might use this code:

```
os_database *db1 ;
. . .
os_List<part*> &a_list =
  *new(db1, os_List<part*>::get_os_typespec())
    os_List<part*>;
part *a_part ;
. . .
a_list |= a_part->children ;
```

This is behaviorally equivalent to

```
part *p ;
os_Cursor<part*> c(a_part->children) ;
for ( p = c.first() ; p ; p = c.next() )
  a_list.insert(p) ;
```

Chapter 3 Using Cursors to Navigate Collections

The ObjectStore collections facility provides a class that helps you navigate within a collection. The `os_Cursor` class helps you insert and remove elements, as well as retrieve particular elements or sequences of elements.

This chapter discusses the following topics:

Description of a Cursor

A *cursor*, an instance of `os_Cursor`, is used to designate a position within a collection. You can use cursors to traverse collections as well as to retrieve, insert, remove, and replace elements.

When you create a default cursor, you specify its associated collection, and the cursor is positioned at the collection's first element. With member functions of `os_Cursor`, you can reposition the cursor as well as retrieve the element at which the cursor is currently positioned. See Traversing Collections with Default Cursors on page 37.

Some members of the collection classes take cursor arguments. These functions support insertion, removal, and replacement of elements.

Creating Default Cursors

The class os_Cursor is a parameterized class supplied by the ObjectStore class library.

```
os_Cursor(const os_Collection<E> &) ;
```

Its constructor takes an os_Collection& (people in the example in the next section) as the argument. This is the collection to be traversed. The traversal proceeds in an arbitrary order for unordered collections and, for ordered collections, in the order in which the elements were inserted.

Note that traversal of a collection with duplicates visits each element once *for each time it occurs* in the collection. For example, an element that occurs three times in a collection is visited three times during a traversal of the collection.

os_Cursor's parameter (person* in the example) indicates the type of elements in the collection being traversed. The cursor's parameter must be the element type (see Using Collections with the Element Type Parameter on page 24) of the collection passed as the constructor argument.

Traversing Collections with Default Cursors

The ObjectStore collections facility allows you to program loops that process the elements of a collection one at a time. When you use it, you do not need to know how many elements are in the collection; each time through the loop, you can test whether more elements remain to be visited. Consequently, you need not worry about loop bounds.

To traverse a collection, you create a cursor associated with the collection you want to traverse. The cursor records the state of an iteration by pointing to the element currently being visited. Each time through the loop, you advance the cursor to the next element and retrieve that element. Following is an example:

```
os_database *db1 ;
. . .
os_Set<person*> &people
  = *new(db1, os_Set<person*>::get_os_typespec())
      os_Set<person*>;
. . . /* insertions into people */

os_Set<person*> &teenagers
  = *new(db1, os_Set<person*>::get_os_typespec())
      os_Set<person*>;

person* p;
os_Cursor<person*> c(people);

for (p = c.first(); c.more() ; p = c.next()) {
  if (p->age >=13 && p->age <= 19)
    teenagers.insert(p);
}
```

The `for` loop in this example retrieves each element of the collection `people` and adds those between the ages of 13 and 19 to the collection `teenagers`. The next sections provide information about positioning and moving cursors in the collection.

Positioning the Cursor at the Collection's First Element

In the example code, the traversal is performed with a `for` loop. The initialization part of the loop header is an assignment involving a call to the member function `os_Cursor::first()`:

```
p = c.first()
```

This positions the cursor at the collection's first element and returns that element. If there is no first element because the collection is empty, `first()` makes the cursor null and returns `0`.

Moving the Cursor to the Collection's Next Element

In the example, the increment part of the `for` loop header is an assignment involving a call to the member function `os_Cursor::next()`:

```
p = c.next()
```

This positions the cursor at the collection's next element and returns that element. If there is no next element, `next()` makes the cursor null and returns `0`.

Is the Cursor at a Nonnull Element?

In the example, the loop's condition is a call to the member function `os_Cursor::more()`:

```
c.more()
```

This function returns a nonzero 32-bit integer (true) when the cursor is still positioned at some element of the collection. The function returns `0` (false) when the cursor is not pointing at any element.

After `next()` is applied to the collection's last element, the cursor becomes null and `more()` then returns `false`, terminating the loop.

Alternative to using more()

For collections that do not allow null elements, you can take advantage of the fact that `first()` and `next()` return null pointers when there is no first or next element. This means you can use the values returned by these functions (in this case `p`) as the loop condition as long as the collection contains no null pointers. For example:

```
os_database *db1 ;
. . .
os_Set<person*> &people
  = *new(db1, os_Set<person*>::get_os_typespec())
      os_Set<person*>;

. . .   /* inserts to people */

os_Set<person*> &teenagers
  = *new(db1, os_Set<person*>::get_os_typespec())
      os_Set<person*>;

person* p ;
os_Cursor<person*> c(people) ;

for ( p = c.first() ; p ; p = c.next() )
  if ( p->age >=13 && p->age <= 19 )
    teenagers.insert(p) ;
```

Rebinding Cursors to Another Collection

You can change a cursor's associated collection with the following members of `os_Cursor`:

```
void rebind(const os_Collection<E>&) ;

void rebind(const os_Collection<E>&, _Rank_fcn) ;
```

This last overloading is for rebinding cursors whose order is specified by a rank function. Once rebound, the cursor is positioned at the specified collection's first element.

Accessing Collection Elements with a Cursor or Ordinal Value

You can gain access to a specific place in a collection by means of an ordinal value or a cursor as arguments to the following functions:

```
void os_Collection::insert_after(const E, const os_Cursor<E>&)
void os_Collection::insert_after(const E, os_unsigned_int32)
void os_Collection::insert_before(const E, const os_Cursor<E>&)
void os_Collection::insert_before(const E, os_unsigned_int32)
void os_Collection::remove_at(const os_Cursor<E>&)
void os_Collection::remove_at(os_unsigned_int32)
E os_Collection::replace_at(const E, const os_Cursor<E>&)
E os_Collection::replace_at(const E, os_unsigned_int32)
E os_Collection::retrieve(const os_Cursor<E>&) const
E os_Collection::retrieve(os_unsigned_int32) const
```

The cursor-based overloadings must use a default cursor. The cursor-based overloadings of `remove_at()`, `replace_at()`, and `retrieve()` can also be used for unordered collections. (See Traversing Collections with Default Cursors on page 37.)

Manipulating First and Last Elements in a Collection

There are also functions for inserting, removing, and retrieving elements from the beginning and the end of an ordered collection. These are declared as follows:

```
void os_Collection::insert_first(const E)
void os_Collection::insert_last(const E)
os_int32 os_Collection::remove_first(const E&)
E os_Collection::remove_first()
os_int32 os_Collection::remove_last(const E&)
E os_Collection::remove_last()
```

The integer-valued `remove()` and `retrieve()` functions return `0` if the collection had no elements to remove or retrieve (that is, was empty). Otherwise, they return a nonzero integer and modify their arguments to indicate the removed or retrieved element.

If you perform any of these functions on an unordered collection created with the supertype's interface, the exception `err_coll_not_supported` is signaled. These operations cause a compile-time error if they are performed on an unordered collection created with the subtype's interface. (Compile-time detection is possible because the unordered subtypes define the ordered operations as `private`.)

Chapter 4
Using Dictionaries

Dictionaries are unordered collections that allow duplicates. Unlike other collections, dictionaries associate a *key* with each element. The key can be a value of any C++ fundamental type, user-defined type, or pointer type. When you insert an element into a dictionary, you specify the key along with the element. You can retrieve an element with a given key or retrieve those elements whose keys fall within a given range.

Required include files

To use ObjectStore’s dictionary facility, you must include the files <os_pse/ostore.hh>, <os_pse/coll.hh>, and either os_pse/coll/dict_pt.hh or <os_pse/coll/dict_pt.cc>, in this order. You must include dict_pt.cc when instantiating the template because it contains the bodies of the functions declared in os_pse/coll/dict_pt.hh. However, users of the template can just include dict_pt.hh.

This chapter discusses the following topics:

Creating Dictionaries

You can create dictionaries with an os_Dictionary constructor. For example:

```
os_Dictionary<char*, char*> *dict = new (
  db, os_Dictionary<char*, char*>::get_os_typespec())
  os_Dictionary<char*, char*>;
```

See the reference information for os_Dictionary::os_Dictionary() on page 123.

Dictionaries can have different types of keys as the key type parameters.

Integer keys

For integer keys, specify one of the following as key type:

- `os_int32` (a signed 32-bit integer)
- `os_unsigned_int32` (an unsigned 32-bit integer)
- `os_int16` (a signed 16-bit integer)
- `os_unsigned_int16` (an unsigned 16-bit integer)

Class keys

For class keys, the class must have a destructor that zeroes any pointers it contains, a default (no arguments) constructor, and `operator=`.

Class keys with soft pointers

For ordered dictionaries and when the key is a class that contains a soft pointer you need to register its assignment operator using the `os_assign_function()` and `os_assign_function_body()` macros. See `os_assign_function()` on page 154 and `os_assign_function_body()` on page 154.

void* keys

Use the type `void*` for pointer keys other than `char*` keys.

char* keys

For `char[]` keys, use the parameterized type `os_char_array<S>`, where the actual parameter is an integer literal indicating the size of the array in bytes.

If a dictionary's key type is `char*`, the dictionary makes its own copies of the character array upon insert. If the dictionary does not allow duplicate keys, you can significantly improve performance by using the type `os_char_star_nocopy` as the key type. With this key type, the dictionary copies the pointer to the array and not the array itself. You can freely pass `chars` to this type.

Note that you should not use `os_char_star_nocopy` with dictionaries that allow duplicate keys.

`char[]`, `char*`, and `os_char_star_nocopy` all use `strcmp()` for comparison.

Floating-point keys

Because the collections library has no defined ordering for IEEE floating-point NaN values, if you are using floating-point keys and expect to generate NaN values, there are a number of possible problems. The collections library will not order floating-point keys with NaN value in any definitive way. All NaN values cannot be expected to be equal. Some may be greater than all the other keys, some may be less than all the other keys. Furthermore cross platform database compatibility is not ensured when using floating-point keys with NaN values; a NaN value generated on one platform cannot be counted on to be equal with a NaN value generated on another platform. This cross-platform problem could result in "lost keys," meaning operations that traverse a dictionary or index might not reach all the keys.

The workaround for this problem is to wrap your floating-point key in a user-defined class and supply your own rank and hash function for that class using the `os_index_key` macro. In this way you can designate the way you want NaN values to be treated and you can address the issue of cross-database compatibility. Keep in mind that these rank and hash functions will be greatly used, so if performance is a priority for your application, you should keep any platform checks and computations to a minimum. See Supplying Rank and Hash Functions on page 58 and `os_assign_function()` on page 154 for more information.

Marking Persistent Dictionaries

If you use persistent dictionaries, you must call the macro `OS_MARK_DICTIONARY()` for each key-type/element-type pair that you use. Calls to this macro have the form

`OS_MARK_DICTIONARY(`*`key-type`*`, `*`element-type`*`)`

Put these calls in the schema source file. For example:

```
/* schema.cc */

#include <os_pse/ostore.hh>
#include <os_pse/coll.hh>
#include <os_pse/coll/dict_pt.hh>
#include <os_pse/manschem.hh>
#include "dnary.hh"
OS_MARK_DICTIONARY(void*,Course*) ;
OS_MARK_DICTIONARY(int,Employee**) ;
OS_MARK_DICTIONARY(int,Course*) ;
OS_MARK_SCHEMA_TYPE(Course) ;
OS_MARK_SCHEMA_TYPE(Employee) ;
OS_MARK_SCHEMA_TYPE(Department) ;
```

For pointer keys, use `void*` as the *`key-type`*.

See the reference information for `OS_MARK_DICTIONARY()` on page 150.

Marking Transient Dictionaries

If you use only transient dictionaries, you must call the macro `OS_TRANSIENT_DICTIONARY()` for each key-type / element-type pair that you use. If you use a particular instantiation of an `os_Dictionary` template both transiently and persistently, you should use the `OS_MARK_DICTIONARY()` macro only. The arguments for `OS_TRANSIENT_DICTIONARY()` are the same as for `OS_MARK_DICTIONARY()`, but you call `OS_TRANSIENT_DICTIONARY()` at file scope in an application source file, rather than in a schema source file.

However, using `OS_TRANSIENT_DICTIONARY()` more than once with the same key type results in a compilation error. For example, the following does not compile correctly:

```
OS_TRANSIENT_DICTIONARY(int,void*);
OS_TRANSIENT_DICTIONARY(int,foo*);
```

The problem is that both invocations of `OS_TRANSIENT_DICTIONARY()` cause a stub routine to be defined for the key type `int`. Instead, you should only invoke `OS_TRANSIENT_DICTIONARY()` once for each key type and use the macro `OS_TRANSIENT_DICTIONARY_NOKEY()` for each consecutive dictionary with the same key type. The correct use, given the example above, would be

```
OS_TRANSIENT_DICTIONARY(int,void*);
OS _TRANSIENT_DICTIONARY_NOKEY(int,foo*);
```

For related information on these macros, see `OS_MARK_DICTIONARY()` on page 150, `OS_TRANSIENT_DICTIONARY()` on page 151, and `OS_TRANSIENT_DICTIONARY_NOKEY()` on page 152.

Dictionary Behavior

Every dictionary has the following properties:

- Duplicate elements are allowed.
- Null pointers cannot be inserted.
- No guarantees are made concerning whether an element inserted or removed during a traversal of its elements will be visited later in that same traversal.

By default, a new dictionary also has the following properties:

- Its entries have no intrinsic order.
- Duplicate keys are allowed; that is, two or more elements can have the same key.
- Range look-ups are not supported; that is, key order is not maintained.

Visiting the Elements with Specified Keys

For dictionaries, you can retrieve an element with the specified key with one of the following two functions:

```
E pick(const K const &key_ref) const ;
E pick(const K *key_ptr) const ;
```

These differ only in that with one you supply a reference to the key, and with the other you supply a pointer to the key. Again, if there is more than one element with the key, an arbitrary one is picked and returned. If there is no such element, the function returns null.

Retrieving Elements When the Key Type is a Class

If the dictionary's key type is a class, you must supply rank and hash functions for the class. See Supplying Rank and Hash Functions on page 58.

The key types `char*`, `char[ ]`, and `os_char_star_nocopy` are each treated as a class whose rank and hash functions are defined in terms of `strcmp()`. For example, for `char*`:

```
a_dictionary.pick("Smith")
```

returns an element of `a_dictionary` whose key is the string "Smith" (that is, whose key, `k`, is such that `strcmp(k, "Smith")` is `0`).

Writing Destructors for Dictionaries

There are circumstances in which a slot in an ObjectStore dictionary can be reused. A slot is used for the first time when the first item is hashed to that slot during an insert. A removal of that item causes the slot to be emptied and marked as previously occupied. A subsequent insert of a key that hashes to that slot can result in the reuse of that slot to hold this new key.

When a key is removed, the destructor for the object of type `K` is run. Because the slot can then be reused, it is necessary for the destructor for the object of type `K` to null any pointers to memory that are freed in the destructor.

Example

Following is an example where type `K` is class `myString`:

```
class myString
{
  private:
    char* theString;
    int len;
}
RMString::RMString(char* theChars)
{
  if (theChars == 0)
    len = 0;
  else
    len = strlen(theChars);
  theString = new(os_segment::of(this),
  os_typespec::get_char(),len + 1)
      char[len+1];
  if (theChars == 0)
    theString[0] = '\0';
  else
    strcpy(theString, theChars);
}
RMString::~RMString()
{
  delete[] theString;
/*******************************************************
  The following line solves the multiple delete problem
*******************************************************/
  theString = 0;

}
```

Failure to include the line `theString = 0;` results in the following error if a slot is reused:

```
Invalid argument to operator delete
```

```
<err-0025-0608>Delete failed. Cannot locate a persistent object at
address 0x5780114 (err_invalid_deletion)
```

Example of Using Dictionaries

A *ternary relationship* is a relationship among three objects, such as "student x got grade y in course z." Dictionaries are often useful in representing ternary relationships. This section contains an example involving the classes `Student`, `Grade`, and `Course`, which allow you to store and retrieve information about who got what grade in what course.

Each `Student` object contains two dictionaries that serve to associate a course with the grade the student got in the course. One dictionary supports look-up of the grade given the course, and the other supports look-up of the courses with a given grade.

Note that the `dnary.cc` example includes `<os_pse/coll/dict_pt.cc>` instead of `dict_pt.hh`; `dict_pt.cc` is needed because it contains the bodies of the functions declared in `dict_pt.hh`. You need not also include `dict_pt.hh` because it is included in `dict_pt.cc.` The example also includes `schema.cc`, which is the application's schema source file.

Following is the file `dnary.hh`, which contains the class definitions. After that is the file `dnary.cc`, which contains the member function implementations.

Header file: dnary.hh

```
/* dnary.hh */

#include <os_pse/ostore.hh>
#include <os_pse/coll.hh>
#include <os_pse/coll/dict_pt.hh>
#include <iostream>

class Student ;
class Grade ;
class Course ;

/* ------------------- class Student ------------------*/
class Student {
  public:
    int get_id() const ;
    const char* get_name() ;

    int add_course( Course*, c, Grade* g = 0 ) ;
    void remove_course(Course*) ;

    Grade *get_grade_for_course(const Course*) const ;
    void set_grade_for_course(Course*, Grade*) ;

    os_Set<Course*> &get_courses_with_grade(
        const Grade* ) const ;
    float get_gpa() const ;

    static os_typespec *get_os_typespec() ;
      { return os_ts<Student>::get(); }

    Student(int id, const char* n);
    ~Student() ;

  private:
    int id ;
    char* name;
    os_Set<Course*> & courses ;
    os_Dictionary<void*, Grade*> * course_grade;
    os_Dictionary<void*, Course*> * grade_course;
```

```
} ;
* --------------------- class grade ---------------------*/
class Grade {

  public:
    const char *get_name() const ;
    float get_value() const ;

    static os_typespec *get_os_typespec() ;
        { return os_ts<Grade>::get(); }

    Grade(const char *name, float value) ;
    ~Grade() ;

  private:
    char *name ;
    float value ;

} ;
* --------------------- class course ---------------------*/
class Course {

  public:
    int get_id() const ;
    const char *name() const ;
    static os_typespec *get_os_typespec() ;
        {return os_ts<Course>::get(); }

    Course(char *name() const ;
    ~Course();

  private:
    int id ;
    char * _name;

} ;
```

Main program: dnary.cc

```
/* dnary.cc */

#include "dnary.hh"
#include <os_pse/coll/dict_pt.cc>

typedef os_Dictionary<void*,Course*> grade_course_dnary ;
typedef os_Dictionary<void*,Grade*> course_grade_dnary ;

/* Student member function implementations */

int Student::get_id() const {
  return id ;
}

const char* Student::get_name() {
  return name ;
}

int Student::add_course( Course *c, Grade *g ) {
  if ( courses->contains(c) )
    return 0 ;

  courses->insert(c) ;
  if (g) {
    grade_course->insert(g, c) ;
    course_grade->insert(c, g) ;
  } /* end if */
  return 1 ;
}
```

```
void Student::remove_course(Course *c) {
  if (courses->contains(c)) {
    courses->remove(c) ;
    grade_course.remove( course_grade->pick(c), c ) ;
    course_grade->remove_value(c) ;
  }
}

Grade *Student::get_grade_for_course(const Course *c) const {
  return course_grade->pick((Course*)c) ;
}

void Student::set_grade_for_course(Course *c, Grade *g) {
  grade_course->remove(course_grade->pick(c), c) ;
  course_grade->remove_value(c) ;
  grade_course->insert(g, c) ;
  course_grade->insert(c, g) ;
}

os_Set<Course*> &
Student::get_courses_with_grade(const Grade *g) const {
  os_Set<Course*> &the_courses =
      *new(os_database::get_transient_database(),
        os_Set<Course*>::get_os_typespec())
        os_Set<Course*> ;

//  os_cursor cur(*grade_course, os_coll_range(
//      os_collection::EQ, g)) ;

  os_cursor cur(*grade_course);

  for ( Course *c = (Course*) cur.first() ; c ; c =
      (Course*) cur.next() )
    the_courses.insert(c) ;

  return the_courses ;
}

float Student::get_gpa() const {
  float sum = 0.0 ;
  os_cursor c(course_grade) ;
  for ( Grade *g = (Grade*) c.first(); g; g = (Grade*) c.next() )
      sum = sum + g->get_value() ;
  return sum / course_grade->cardinality();
}

Student::Student(int i, const char* n) :

    courses = new(os_segment::of(this),
          os_Set<Course*>::get_os_typespec())
      os_Set<Course*>,

    course_grade = new(os_segment::of(this),
      os_Dictionary<void*, Grade*>::get_os_typespec())
      os_Dictionary<void*, Grade*>(10);
    grade_course = new (os_segment::of(this),
      os_Dictionary<void*, Course*>::get_os_typespec())
      os_Dictionary<void*, Course*> () ;
  id = i;
  if(n) {
    int len + ::strlen(n) + 1;
    _name = new(os_segment::of(this), os_typespec::get_char(), len)
        char[len];
    ::strcpy(_name, n);
```

```
  } else
    _name = 0;
}
Student::~Student() {
  if (_name)
      delete {} _name;
  delete courses ;
  delete course_grade ;
  delete grade_course ;
}
/* Grade member function implementations */
const char *Grade::get_name() const {
  return name ;
}
float Grade::get_value() const {
  return value ;
}
Grade::Grade(const char *n, float v) {
  name = new( os_segment::of(this), os_typespec::get_char(),
    strlen(n)+1 )
char[strlen(n)+1] ;
  ::strcpy(name, n) ;
  value = v ;
}
Grade::~Grade() {
  delete name ;
}
/* Course member function implementations */
int Course::get_id() const {
  return id ;
}
  const char * Course::name() const{
    return _name;
}
Course::Course(char* name, int _id)
  : id(_id)
{
  int len = ::strlen(name) + 1;
  _name = new(os_segment::of(this), os_typespec::get_char(), len)
      char[len];
  ::strcpy(_name, name);
}
Course::~Course()
{
  delete [] _name;
}
void force_vfts(void *) {
  force_vfts(new os_Dictionary<char *, Grade *>);
  force_vfts(new os_Dictionary<os_char_star_nocopy, Course *>);
  force_vfts(new os_Dictionary<void *, Course *>);
  force_vfts(new os_Dictionary<void *, Grade *>);
  force_vfts(new  os_List<Student *>);
  force_vfts(new  os_List<Course *>);
```

```
  force_vfts(new  os_Set<Student *>);
  force_vfts(new  os_Set<Course *>);
  force_vfts(new  os_Array<Course *>);
  force_vfts(new  os_Array<Student *>);
  force_vfts(new  os_Bag<Student *>);
  force_vfts(new  os_Bag<Course *>);
  force_vfts(new  os_Collection<Student *>);
}
```

Schema file: schema.cc

```
/* schema.cc */

#include <os_pse/ostore.hh>
#include <os_pse/coll.hh>
#include <os_pse/coll/dict_pt.hh>
#include <os_pse/manschem.hh>

#include "dnary.hh"

  OS_MARK_DICTIONARY(void*,Course*);
  OS_MARK_DICTIONARY(void*,Grade*);
  OS_MARK_SCHEMA_TYPE(Course);
  OS_MARK_SCHEMA_TYPE(Student);
  OS_MARK_SCHEMA_TYPE(Grade);
```

Example description

The data member `Student::courses` contains a reference to a set of pointers to the courses the student has taken.

The data member `Student::course_grade` contains a reference to a dictionary that maps each course to the grade the student got for that course. This dictionary supports look-up of the grade given the course.

The data member `Student::grade_course` contains a reference to a dictionary that maps each grade to the courses for which the student got that grade. This dictionary supports look-up of the courses given the grade.

The function `Student::add_course()` first checks to see if the specified course has already been added. If it has, `0` (indicating failure) is returned. If it has not, the function inserts the specified course into the set referred to by `Student::courses`. Then, if a grade is specified, entries are inserted into the dictionaries referred to by `Student::course_grade` and `Student::grade_course`. Finally, `1` (indicating success) is returned.

The function `Student::remove_course()` removes the specified course from `Student::courses`. If the course is not an element of `courses`, the call to `remove()` has no effect.

Then, using `pick()` on `course_grade`, `remove_course()` determines the grade for the given course. The grade and the course are then passed to `os_Dictionary::remove()` to remove from `Student::grade_course` the entry whose value is the given course. The function `add_course()` ensures that there is at most one.

If the course is not an element of the dictionary, `pick()` returns `0` and the call to `remove()` has no effect.

Finally, using `os_Dictionary::remove_value()`, `remove_course()` removes from `Student::Course_grade` the entry whose key is the given course. Again, `add_` `course()` ensures there is at most one. If the dictionary has no entry whose key is that course, the call to `remove_value()` has no effect.

`Student::get_grade_for_course()` uses `os_Dictionary::pick()` to retrieve from `Student::course_grade` the element whose key is the given course.

`Student::set_grade_for_course()` first takes precautions in case the specified course already has been assigned a grade. It removes from `Student::course_grade` and `Student::grade_course` any entries with the given course. It does this as follows.

First, the function performs `remove()` on `grade_course`, passing in the grade for the given course (determined by performing `pick()` on `course_grade`) and also passing in the given course itself. If no grade has been set for the course, `pick()` returns `0` and the call to `remove()` has no effect. Then the function uses `remove_value()` to remove from `course_grade` the entry, if there is one, whose key is the given course.

Next, `Student::set_grade_for_course()` inserts into `grade_course` an entry whose key is the specified grade and whose value is the specified course. Finally, it inserts into `course_grade` an entry whose key is the specified course and whose value is the specified grade.

The function `Student::get_courses_with_grade()` returns a reference to a collection of the courses for which the student got the specified grade. It creates a collection on the transient heap and then uses a restricted cursor to visit each element of `grade_course` whose key is the specified grade. As each qualifying element is visited, it is inserted into the newly created collection. Finally, a reference to the collection is returned.

The function `Student::get_gpa()` returns the student's grade point average. It visits each element of the dictionary `course_grade`, summing the result of performing `get_value()` on each element along the way. When the traversal is complete, the sum is divided by the dictionary's size to get the average, which is returned.

The `Student` constructor allocates an `os_Set` and two instances of `os_Dictionary` in the specified segment.

Chapter 5
Performing Advanced Collections Operations

After you are familiar with the basic use of collections, you can use the information in this chapter to perform more advanced operations with collections. This chapter discusses the following topics:

Controlling Traversal Order

To control traversal order, use one of the constructors for `os_Cursor`. The various overloadings allow you to specify a traversal order based on

- The order in persistent memory of the objects pointed to by collection elements
- The rank function registered for the collection's element type
- A specified rank function

The following sections discuss traversals.

Default Traversal Order

```
os_Cursor( const os_Collection&, os_int32 options );
```

Every cursor has an associated ordering for the elements of its associated collection. By default, this ordering is the order in which each element appears in the collection (for ordered collections) or an arbitrary ordering (for unordered collections).

Address Order Traversal

If you supply `os_collection::order_by_address` as the `options` argument, this cursor iterates in address order. This is the order in which the objects pointed to by collection elements are arranged in persistent memory.

If you dereference each collection element as you retrieve it and the objects pointed to by collection elements will not all fit in the client cache at once, this order can dramatically reduce paging overhead.

An order-by-address cursor is update insensitive.

Rank-Function-Based Traversal

```
os_Cursor(
  const os_Collection&,
  const char* typename,
) ;
```

If you create a cursor with this constructor, iteration follows the order determined by the rank function of the element type specified by `typename`. See Supplying Rank and Hash Functions on page 58.

```
os_Cursor(
  const os_Collection&,
  _Rank_fcn
) ;
```

`_Rank_fcn` is a pointer to a rank function for the element type. Iteration using a cursor created with this constructor follows the order determined by this function.

Rank-function-based cursors are update insensitive. See Performing Collection Updates During Traversal on page 55.

Performing Collection Updates During Traversal

If you want to be able to update a collection while traversing it, you must use an *update-insensitive* cursor.

With an update-insensitive cursor, the traversal is based on a snapshot of the collection elements at the time the cursor was bound to the collection. None of the inserts and removes performed on the collection is reflected in the traversal.

If you update a collection while traversing it without using an update-insensitive cursor, the results of the traversal are undefined.

You can create an update-insensitive cursor with the following cursor constructor:

```
os_Cursor(
const os_Collection&,
os_int32 options
) ;
```

Supply `os_collection::update_insensitive` as the *`options`* argument.

In addition, the following kinds of cursors are always update insensitive:

- Rank-function-based cursors. See Rank-Function-Based Traversal on page 54.
- `order_by_address` cursors. See Address Order Traversal on page 54.

Retrieving Uniquely Specified Collection Elements

You can retrieve the collection element at which a specified cursor is positioned with the following function:

```
E retrieve(const os_Cursor<E>&) const ;
```

If the cursor is null, `err_coll_null_cursor` is signaled. If the cursor is nonnull but not positioned at an element, `err_coll_illegal_cursor` is signaled.

You can retrieve the only element of a collection with the following function:

```
E only() const ;
```

If the collection has more than one element, `err_coll_not_singleton` is signaled. If the collection is empty, `0` is returned.

Ordered Collections

For ordered collections, you can retrieve the element with a specified numerical position with the following function:

```
E retrieve(os_unsigned_int32 index) const ;
```

The index is zero based. If the index is not less than the collection's size, `err_coll_out_of_range` is signaled. If the collection does not have `maintain_order` behavior, `err_coll_not_supported` is signaled.

retrieve_first() function

You can retrieve a collection's first element with the following function:

```
E retrieve_first() const ;
```

This function returns the collection's first element, or `0` if the collection is empty. If the collection is not ordered, `err_coll_not_supported` is signaled.

For collections with `allow_nulls` behavior, you can use the following function instead:

```
os_int32 retrieve_first(const E&) const ;
```

This function modifies the argument to refer to the collection's first element. It returns `0` if the specified collection is empty, and nonzero otherwise. If the collection is not ordered, `err_coll_not_supported` is signaled.

retrieve_last() function

To retrieve a collection's last element, use

```
E retrieve_last() const ;
```

This function returns the collection's last element, or `0` if the collection is empty. If the collection is not ordered, `err_coll_not_supported` is signaled.

For collections with `allow_nulls` behavior, you can use the following function instead:

```
os_int32 retrieve_last(const E&) const ;
```

This function modifies the argument to refer to the collection's last element. It returns `0` if the specified collection is empty, and nonzero otherwise. If the collection is not ordered, `err_coll_not_supported` is signaled.

Selecting Individual Collection Elements with pick()

Dictionaries

For dictionaries, you can retrieve an element with the specified key, with one of the following two functions:

```
E pick(const K const &key_ref) const ;
```

```
E pick(const K *key_ptr) const ;
```

These two differ only in that with one you supply a reference to the key, and with the other you supply a pointer to the key. Again, if there is more than one element with the key, an arbitrary one is picked and returned. If there is no such element, `0` is returned.

If the dictionary's key type is a class, you must supply rank and hash functions for the class (see Supplying Rank and Hash Functions on page 58).

The key types `char*`, `char[ ]`, and `os_char_star_nocopy` are each treated as a class whose rank and hash functions are defined in terms of `strcmp()`. For example, for `char*`:

```
a_dictionary.pick("Smith")
```

returns an element of `a_dictionary` whose key is the string `Smith` (that is, whose key, `k`, is such that `strcmp(k, "Smith")` is `0`).

Picking an Arbitrary Element

You can retrieve an arbitrary collection element with

```
E pick() const;
```

If the collection is empty, `0` is returned.

This is sometimes useful when all the elements of a collection have the same value for a data member and the easiest way to retrieve this value is through one of the elements.

For example, suppose the class `bus` defines a member for the set of pins connected to it but no member for the cell in which it resides, while `pin` defines a member pointing to its attached cell, which in turn has a member pointing to its containing cell. The best way to find the cell on which a given bus resides is to find the pins connected to it and then find the cell on which one of the pins resides:

```
a_cell = a_bus–>pins.pick()–>cell->container;
```

Consolidating Duplicates with operator =()

You can use the assignment operator `os_Collection::operator =()` (see Copying, Combining, and Comparing Collections on page 33) to consolidate duplicates in a bag or other collection. Do this by assigning the collection with duplicates to an empty collection that does not allow duplicates. For example:

```
os_database *db1 ;
part *a_part, *p ;
employee *e ;
. . .
os_Bag<employee*> &emp_bag =
    *new(db1, os_Bag<employee*>::get_os_typespec())
    os_Bag<employee*>;

os_Set<employee*> &emp_set =
  = *new(db1, os_Set<employee*>::get_os_typespec())
    os_Set<employee*>;

os_Cursor<part*> c(a_part->children) ;
for ( p = c.first() ; p ; p = c.next() )
  emp_bag.insert(p->responsible_engineer) ;

emp_set = emp_bag ;   // consolidate duplicates

os_Cursor<employee*> c(emp_set) ;

for (e = c.first() ; e ; e = c.next() )
  cout << e->name << "\t" << emp_bag.count(e) << "\n" ;
```

If two of `a_part`'s children have the same `responsible_engineer`, that engineer appears twice as an element of `emp_bag`. You can consolidate duplicates in `emp_set` so you can iterate over it, retrieving each engineer only once in the loop and then use `count()` to see how many times the engineer occurs in `emp_bag`. This is the number of parts for which the engineer is responsible.

Supplying Rank and Hash Functions

In all these examples, iteration order is based on integer-valued data members (`part_number`, `emp_id`, or `salary`); that is, the paths end in integer values. The integers have a system-supplied order, defined by the comparison operators <, >, and so forth. The same is true for pointers. For `char*` pointers, which are treated differently from other pointers, the order is defined by performing `strcmp()` on the string pointed to. But what if a path ends in some other type of value; that is, what if it ends in the instances of some class or floating-point numerical type?

If you want to use such a path to control iteration order, you must make known to ObjectStore a utility specific to the class, a *rank* function that defines an ordering on the type's instances.

Registering

To register a rank or hash function, you must call `os_index_key()` from within a session. This function registers a rank or hash function for the entire process. Calling `os_index_key()` outside any session has no effect.

Floating-point keys

If you are using floating-point keys and expect to generate NaN values, you should wrap your floating-point key in a user-defined class and supply your own rank and hash function for that class using the os_index_key macro.

os_index_key() Macro

You make these utilities known to ObjectStore by calling the macro `os_index_key()`. Calls to `os_index_key()` have the following form:

```
os_index_key(type,rank-function,hash-function);
```

For example:

```
os_index_key(date,date_rank,date_hash);
```

The *`type`* is the type that is at the end of the path.

For information about the `os_index_key()` macro, see `os_assign_function()` on page 154.

The *`rank-function`* is a user-defined global function that, for any pair of instances of *`class`*, provides an ordering indicator for the instances, much as `strcmp()` does for strings. The rank function should return one of `os_collection::LT`, `os_collection::GT`, or `os_collection::EQ`.

Rank functions for floating-point numerical types (`float`, `double`, and `long double`) should follow these guidelines:

- `NaN` and `inf` must be handled specially. For example, the representation of `NaN` is not unique. In the rank function, test for these values before doing anything else. You might want `NaN` to rank below any other value.

For the purpose of ranking, comparisons should be precise. For example, the rank function should consider `x` and `y` to be equal if `x == y` but not if `abs(x – y) < e` (for some small value of `e`) as long as `>` and `<` also check for equality using `e`.The *`hash-function`* is a user-defined global function that, for each instance of *`class`*, returns a value, an `os_unsigned_int32`, that can be used as a key in a hash table. It takes a `const void*` argument. If you are not supplying a hash function for the class, this argument should be `0`.

Supposethat dates are instances of a user-defined class:

```
class date{
public:
  int month;
  int day;
  int year;
};
```

In this case, you must define the rank function and make it known to ObjectStore. You might define it as follows:

```
int date_rank(const void* arg1, const void *arg2) {
  const date *date1 = (const date *) arg1;
  const date *date2 = (const date *) arg2;

  if (date1->year < date2->year)
    return os_collection::LT;
  else if (date1->year > date2->year)
    return os_collection::GT;
  else if (date1->month < date2->month)
    return os_collection::LT;
  else if (date1->month > date2->month)
    return os_collection::GT;
  else if (date1->day < date2->day)
    return os_collection::LT;
  else if (date1->day > date2->day)
    return os_collection::GT;
  return os_collection::EQ;
}
```

Specifying Expected Size

Frequently, a collection has a loading phase in which it is loaded with elements before being the subject of other kinds of manipulation such as queries and traversal. In these cases, it is desirable to create the collection with the size it will have after loading is complete.

To presize a collection, use the `expected_size` argument to a collection’s constructor.

Chapter 6
Class Reference

This chapter describes the following classes and their functions and enumerators. See the Introduction on page 61 for information that applies to all classes.

Introduction

The following information applies to all classes related to collections:

Type definitions

The types `os_int32` and `os_boolean`, used throughout this manual, are each defined as a signed 32-bit integer type. The type `os_unsigned_int32` is defined as an unsigned 32-bit integer type.

Required header files

Programs that use collections or collection subtypes (arrays, bags, dictionaries, lists, and sets), after they include the header file <os_pse/ostore.hh>, must include the header file <os_pse/coll.hh>.

os_Array

```
template <class E>
class os_Array : public os_Collection<E>
```

An array, like a list (see os_List on page 127), is an ordered collection. Arrays always provide access to collection elements in constant time. That is, the time complexity of operations such as retrieval of the n^{th} element is order 1 in the array's cardinality.

Arrays also have a `set_cardinality()` function that changes the array cardinality, filling the additional array slots (if the cardinality is increased) with a specified fill value. Arrays allow both duplicates and nulls. As with other ordered collections, array elements can be inserted, removed, replaced, or retrieved based on a specified numerical array index or based on the position of a specified cursor.

If an element is inserted into an `os_Array`, elements after it are pushed down in order. If an element is removed, elements after it in the array are pushed up. If you want the index to an element to remain constant, set the element at index*n* to either `0` or another pointer.

The class `os_Array` is parameterized, with a parameter for constraining the type of values allowable as elements (for the nonparameterized version of this class, see `os_array` on page 68). The element type parameter, `E`, occurs in the signatures of some of the functions described here. The parameter is used by the compiler to detect type errors.

The element type of any collection type, such as an array, must be a pointer type (for example, `employee*`).

Create `os_Array`s with the os_Array constructor.

You must mark parameterized collections types in the schema source file.

Member functions and enumerators

The following tables list the member functions that can be performed on instances of `os_Array`. The second table lists the enumerators inherited by `os_Array` from `os_collection`. Many functions are inherited by `os_Array` from `os_Collection or os_collection`. The full explanation of each inherited function or enumerator appears in the entry for the class from which it is inherited. The full explanation of each function defined by `os_Array` appears in this entry, after the table and list. In each case, the *Defined By* column gives the class whose entry contains the full explanation.

In the following table, for parameterized `os_Array<E>, E` means class pointer.

Name	*Arguments*	*Returns*	*Defined By*
`cardinality`	`( ) const`	`os_int32`	`os_collection`
`clear`	`( )`	`void`	`os_collection`
`contains`	`( const E ) const`	`os_int32`	`os_Collection`
`count`	`( const E ) const`	`os_int32`	`os_Collection`
`empty`	`( )`	`os_int32`	`os_collection`

Name	*Arguments*	*Returns*	*Defined By*
get_behavior	( ) const	os_unsigned_ int32	os_collection
insert	(const E)	void	os_Collection
insert_after	(const E, const os_Cursor<E>&)	void	os_Collection
	(const E, os_unsigned_int32)	void	
insert_before	(const E, const os_Cursor<E>&)	void	os_Collection
	(const E, os_unsigned_int32)	void	
insert_first	(const E)	void	os_Collection
insert_last	(const E)	void	os_Collection
only	( ) const	E	os_Collection
operator os_array&	( )		os_collection
operator const os_array&	( ) const		os_collection
operator os_Bag<E>&	( )		os_Collection
operator const os_Bag<E>&	( ) const		os_Collection
operator os_bag&	( )		os_collection
operator const os_bag&	( ) const		os_collection
operator os_List<E>&	( )		os_Collection
operator const os_List<E>&	( ) const		os_Collection
operator os_list&	( )		os_collection
operator const os_list&	( ) const		os_collection
operator os_Set<E>&	( )		os_Collection
operator const os_Set<E>&	( ) const		os_Collection
operator os_set&	( )		os_collection
operator const os_set&	( ) const		os_collection
operator ==	(const os_Collection<E>&) const	os_int32	os_Collection
	(E) const	os_int32	
operator !=	(const os_Collection<E>&) const	os_int32	os_Collection
	(E) const	os_int32	
operator <	(const os_Collection<E>&) const	os_int32	os_Collection
	(E) const	os_int32	

Name	*Arguments*	*Returns*	*Defined By*
operator <=	(const os_Collection<E>&) const (E) const	os_int32 os_int32	os_Collection
operator >	(const os_Collection<E>&) const (E) const	os_int32 os_int32	os_Collection
operator >=	(const os_Collection<E>&) const (E) const	os_int32 os_int32	os_Collection
operator =	(const os_Array<E>&) const (const os_Collection<E>&) const (E) const	os_Array<E>& os_array& os_array	os_Array
operator |=	(const os_Collection<E>&) const (E) const	os_Array<E>& os_Array<E>&	os_Array
operator |	(const os_Collection<E>&) const (E) const	os_Collection<E>& os_Collection<E>&	os_Collection
operator &=	(const os_Collection<E>&) const (E) const	os_Array<E>& os_Array<E>&	os_Array
operator &	(const os_Collection<E>&) const (E) const	os_Collection<E>& os_Collection<E>&	os_Collection
operator –=	(const os_Collection<E>&) const (E) const	os_Array<E>& os_Array<E>&	os_Array
operator -	(const os_Collection<E>&) const (E) const	os_Collection<E>& os_Collection<E>&	os_Collection
os_Array<E>	( ) (os_unsigned_int32 expected_size) (const os_Array<E>&) (const const os_Collection<E>&) (os_unsigned_int32 expected_size = 0, os_unsigned_int32 card = 0, E fill_value)		os_Array
remove	(const E)	os_int32	os_Collection

Name	*Arguments*	*Returns*	*Defined By*
remove_at	(const os_Cursor<E>&) (os_unsigned_int32)	void void	os_Collection
remove_first	(const E&) ( )	os_int32 E	os_Collection
remove_last	(const E&) ( )	os_int32 E	os_Collection
replace_at	(const E, const os_Cursor<E>&) (const E, os_unsigned_int32)	E E	os_Collection
retrieve	(os_unsigned_int32) const (const os_Cursor<E>&) const	E E	os_Collection
retrieve_first	( ) const (const E&) const	E os_int32	os_Collection
retrieve_last	( ) const (const E&) const	E os_int32	os_Collection
set_cardinality	(os_unsigned_int32 new_card, E fill_value)	void	os_Array

os_Array enumerators

The following table lists the enumerators that can be used for `os_Array` member functions.

Name	*Inherited From*
allow_duplicates	os_collection
allow_nulls	os_collection
EQ	os_collection
GT	os_collection
LT	os_collection
maintain_order	os_collection

Assignment Operator Semantics

Assignment operator semantics are described for the following functions in terms of insert operations into the target collection. Although the actual implementation of the assignment might be different, the associated semantics are maintained.

os_Array::operator =()

```
os_Array<E> &operator =(const os_Array<E> &s);
```

```
os_Array<E> &operator =(const os_Collection<E> &s);
```

Copies the contents of the collection *s* into the target array and returns the target array. The copy is performed by effectively clearing the target, iterating over the source collection, and inserting each element into the target array. The iteration is ordered if the source collection is ordered. The target array semantics are enforced as usual during the insertion process.

```
os_Array<E> &operator =(const E e);
```

Clears the target array, inserts the element *e* into the target array, and returns the target array.

os_Array::operator |=()

```
os_Array<E> &operator |=(const os_Collection<E> &s);
```

Inserts the elements contained in *s* into the target array and returns the target array. In effect, this appends the elements of a collection to an `os_Array`.

```
os_Array<E> &operator |=(const E e);
```

Inserts the element *e* into the target array and returns the target array.

os_Array::operator &=()

```
os_Array<E> &operator &=(const os_Collection<E> &s);
```

For each element in the target collection, reduces the count of the element in the target to the minimum of the counts in the source and target collections. If the collection is ordered and contains duplicates, it does so by retaining the appropriate number of leading elements. The function returns the target collection.

```
os_Array<E> &operator &=(const E e);
```

If *e* is present in the target, converts the target into a collection containing just the element *e*. Otherwise, it clears the target collection. The function returns the target collection.

os_Array::operator –=()

```
os_Array<E> &operator -=(const os_Collection<E> &s);
```

For each element in the collection *s*, removes `s.count(e)` occurrences of the element from the target collection. If the collection is ordered, it is the first `s.count(e)` elements that are removed. The function returns the target collection.

```
os_Array<E> &operator -=(const E e);
```

Removes the element *e* from the target collection. If the collection is ordered, it is the first occurrence of the element that is removed from the target collection. The function returns the target collection.

os_Array::os_Array()

```
os_Array();
```

Returns an empty array.

```
os_Array(os_unsigned_int32);
```

Returns an empty array whose initial implementation is based on the expectation that the specified `os_unsigned_int32` indicates the approximate usual cardinality of the array after it has been loaded with elements.

```
os_Array(const os_Array<E>&);
```

Returns an array that results from assigning the specified array to an empty array.

```
os_Array(const os_Collection<E>&);
```

Returns an array that results from assigning the specified collection to an empty array.

```
os_Array(os_unsigned_int32 expected_size = 0,
  os_unsigned_int32 card = 0,
  void* fill_value = 0 );
```

The arguments have the following meaning:

- *`expected_size`* presizes the array. If this argument is not specified, the array is presized to 5 slots.
- *`card`* specifies the cardinality of the initial array.
- The array's slots are set to the value specified by *`fill_value`*. If this argument is unspecified, the array is null filled.

os_Array::set_cardinality()

```
void set_cardinality(os_unsigned_int32 new_card, E fill_value);
```

Augments the array to have the specified cardinality, using the specified *`fill_value`* to occupy the array's new slots.

os_array

`class os_array : public os_collection`

An array, like a list (see the class `os_list` on page 132), is an ordered collection. Unlike lists, arrays allow nulls and have a `set_cardinality( )` function that changes the array cardinality, filling the additional array slots (if the cardinality is increased) with a specified fill value.

Arrays allow both duplicates and nulls. Array elements can be inserted, removed, replaced, or retrieved based on a specified numerical array index or based on the position of a specified cursor.

If an element is inserted into an `os_array`, elements after it are pushed down in order. If an element is removed, elements after it in the array are pushed up. If you want the index to an element to remain constant, set the element at index*n* to either `0` or another pointer.

The class `os_array` is nonparameterized. For the parameterized version of this class, see `os_Array` on page 62.

Array elements are pointers, so the element type of any array must be a pointer type (for example, `employee*`).

Create arrays with the os_array constructor.

Tables of member functions and enumerators

The first of the following tables lists the member functions that can be performed on instances of `os_array`. The second table lists the enumerators inherited by `os_array` from `os_collection`. Many functions are also inherited by `os_array` from `os_collection`. The full explanation of each inherited function or enumerator appears in the entry for the class from which it is inherited. The full explanation of each function defined by `os_array` appears in this entry, after the tables. In each case, the *Defined By* column gives the class whose entry contains the full explanation.

Name	*Arguments*	*Returns*	*Defined By*
cardinality	( ) const	os_int32	os_collection
clear	( )	void	os_collection
contains	(const void*) const	os_int32	os_collection
count	(const void*) const	os_int32	os_collection
empty	( ) const	os_int32	os_collection
get_behavior	( ) const	os_unsigned_ int32	os_collection
insert	(const void*)	void	os_collection
insert_after	(const void*, const os_cursor&) (const void*, os_unsigned_int32)	void void	os_collection

<table>
<tr><th>Name</th><th>Arguments</th><th>Returns</th><th>Defined By</th></tr>
<tr><td>insert_before</td><td>(const void*,
const os_cursor&)
(const void*,
os_unsigned_int32)</td><td>void

void</td><td>os_collection</td></tr>
<tr><td>insert_first</td><td>(const void*)</td><td>void</td><td>os_collection</td></tr>
<tr><td>insert_last</td><td>(const void*)</td><td>void</td><td>os_collection</td></tr>
<tr><td>only</td><td>( ) const</td><td>void*</td><td>os_collection</td></tr>
<tr><td>operator os_bag&</td><td>( )</td><td></td><td>os_collection</td></tr>
<tr><td>operator const os_bag&</td><td>( ) const</td><td></td><td>os_collection</td></tr>
<tr><td>operator os_list&</td><td>( )</td><td></td><td>os_collection</td></tr>
<tr><td>operator const os_list&</td><td>( ) const</td><td></td><td>os_collection</td></tr>
<tr><td>operator os_set&</td><td>( )</td><td></td><td>os_collection</td></tr>
<tr><td>operator const os_set&</td><td>( ) const</td><td></td><td>os_collection</td></tr>
<tr><td>operator ==</td><td>(const os_collection&) const
(const void*) const</td><td>os_int32
os_int32</td><td>os_collection</td></tr>
<tr><td>operator !=</td><td>(const os_collection&) const
(const void*) const</td><td>os_int32
os_int32</td><td>os_collection</td></tr>
<tr><td>operator <</td><td>(const os_collection&) const
(const void*) const</td><td>os_int32
os_int32</td><td>os_collection</td></tr>
<tr><td>operator <=</td><td>(const os_collection&) const
(const void*) const</td><td>os_int32
os_int32</td><td>os_collection</td></tr>
<tr><td>operator ></td><td>(const os_collection&) const
(const void*) const</td><td>os_int32
os_int32</td><td>os_collection</td></tr>
<tr><td>operator >=</td><td>(const os_collection&) const
(const void*) const</td><td>os_int32
os_int32</td><td>os_collection</td></tr>
<tr><td>operator =</td><td>(const os_array&) const
(const os_collection&) const
(const void*) const</td><td>os_array&
os_array&
os_array</td><td>os_array</td></tr>
<tr><td>operator |=</td><td>(const os_collection&) const
(const void*) const</td><td>os_array&
os_array&</td><td>os_array</td></tr>
<tr><td>operator |</td><td>(const os_collection&) const
(const void*) const</td><td>os_collection&
os_collection&</td><td>os_collection</td></tr>
<tr><td>operator &=</td><td>(const os_collection&) const
(const void*) const</td><td>os_array&
os_array&</td><td>os_array</td></tr>
<tr><td>operator &</td><td>(const os_collection&) const
(const void*) const</td><td>os_collection&
os_collection&</td><td>os_collection</td></tr>
</table>

Name	*Arguments*	*Returns*	*Defined By*
operator -=	(const os_collection&) const (const void*) const	os_array& os_array&	os_array
operator -	(const os_collection&) const (const void*) const	os_collection& os_collection&	os_collection
os_array	( ) (os_unsigned_int32 expected_size) (const os_array&) (const os_collection&) (os_unsigned_int32 expected_size = 0, os_unsigned_int32 card = 0, void *fill_value = 0)		os_array
remove	(const void*)	os_int32	os_collection
remove_at	(const os_cursor&) (os_unsigned_int32)	void void	os_collection
remove_first	(const void*&) ( )	os_int32 void*	os_collection
remove_last	(const void*&) ( )	os_int32 void*	os_collection
replace_at	(const void*, const os_cursor&) (const void*, os_unsigned_int32)	void* void*	os_collection
retrieve	(os_unsigned_int32) const (const os_cursor&) const	void* void*	os_collection
retrieve_first	( ) const (const void*&) const	void* os_int32	os_collection
retrieve_last	( ) const (const void*&) const	void* os_int32	os_collection
set_cardinality	(os_unsigned_int32 new_card, void *fill_value)	void	os_array

os_array enumerators

The following table lists the enumerators that can be used by `os_array` member functions.

Name	*Inherited From*
allow_nulls	os_collection
EQ	os_collection
GT	os_collection

Name	*Inherited From*
`LT`	`os_collection`
`maintain_order`	`os_collection`

Assignment Operator Semantics

Assignment operator semantics are described for the following functions in terms of insert operations into the target collection. The actual implementation of the assignment might be different, but the associated semantics are maintained.

os_array::operator =()

`os_array &operator =(const os_array &s);`

`os_array &operator =(const os_collection &s);`

Copies the contents of the collection *s* into the target collection and returns the target collection. The copy is performed by effectively clearing the target, iterating over the source collection, and inserting each element into the target collection. The iteration is ordered if the source collection is ordered. The target array semantics are enforced as usual during the insertion process.

`os_array &operator =(const void *e);`

Clears the target array, inserts the element *e* into the target array, and returns the target array.

os_array::operator |=()

`os_array &operator |=(const os_collection &s);`

Inserts the elements contained in *s* into the target collection and returns the target collection.

`os_array &operator |=(const void *e);`

Inserts the element *e* into the target collection and returns the target collection. In effect, this appends the elements of a collection to an `os_array`.

os_array::operator &=()

`os_array &operator &=(const os_collection &s);`

For each element in the target collection, reduces the count of the element in the target to the minimum of the counts in the source and target collections. If the collection is ordered and contains duplicates, it does so by retaining the appropriate number of leading elements. The function returns the target collection.

`os_array &operator &=(const void *e);`

If *e* is present in the target, converts the target into a collection containing just the element *e*. Otherwise, it clears the target collection. The function returns the target collection.

os_array::operator –=()

```
os_array &operator –=(const os_collection &s);
```

For each element in the collection *s*, removes `s.count(e)` occurrences of the element from the target collection. If the collection is ordered, it is the first `s.count(e)` elements that are removed. The function returns the target collection.

```
os_array &operator –=(const void *e);
```

Removes the element *e* from the target collection. If the collection is ordered, it is the first occurrence of the element that is removed from the target collection. The function returns the target collection.

os_array::os_array()

```
os_array();
```

Returns an empty array.

```
os_array(os_unsigned_int32);
```

Returns an empty array whose initial implementation is based on the expectation that the specified `os_unsigned_int32` indicates the approximate usual cardinality of the array after it has been loaded with elements.

```
os_array(const os_array&);
```

Returns an array that results from assigning the specified array to an empty array.

```
os_array(const os_collection&);
```

Returns an array that results from assigning the specified collection to an empty array.

```
os_array(os_unsigned_int32 expected_size = 0,
  os_unsigned_int32 card = 0,
  void* fill_value = 0 );
```

The arguments have the following meaning:

- *expected_size* presizes the array. If this argument is not specified, the array is presized to 5 slots.
- *card* specifies the cardinality of the initial array.
- The array's slots are set to the value specified by *fill_value*. If this argument is unspecified, the array is null filled.

os_array::set_cardinality()

```
void set_cardinality(os_unsigned_int32 new_card, void *fill_value);
```

Augments the array to have the specified cardinality, using the specified `fill_value` to occupy the array's new slots.

os_Bag

```
template <class E>
class os_Bag : public os_Collection<E>
```

A bag (sometimes called a *multiset*) is an unordered collection. Unlike elements in sets, elements can occur in a bag more than once at a given time.

The *count* of a value in a given bag is the number of times it occurs in the bag. Each insertion of a value into a bag increases its count in the bag by 1. The count of a value in a bag is 0 if and only if the value is not an element of the bag.

Values are inserted into an `os_Bag` anywhere. That is, the user has no control over the ordering of the elements.

Use the constructor `os_Bag::os_Bag()` to create bags.

You can presize a bag when you create it.

The class `os_Bag` is *parameterized*, with a parameter for constraining the type of values allowable as elements (for the nonparameterized version of this class, see `os_bag` on page 78). The element type parameter, `E`, occurs in the signatures of some of the functions described below. The parameter is used by the compiler to detect type errors.

The element type of any instance of `os_Bag` must be a pointer type.

You must mark parameterized collections types in the schema source file.

Tables of member functions and enumerators

The first of the following tables lists the member functions that can be performed on instances of `os_Bag`. The second table lists the enumerators inherited by `os_Bag` from `os_collection`. Many functions are also inherited by `os_Bag` from `os_Collection` or `os_collection`. The full explanation of each inherited function or enumerator appears in the entry for the class from which it is inherited. The full explanation of each function defined by `os_Bag` appears in this entry, after the tables. In each case, the *Defined By* column gives the class whose entry contains the full explanation.

Name	*Arguments*	*Returns*	*Defined By*
`cardinality`	`( ) const`	`os_int32`	`os_collection`
`clear`	`( )`	`void`	`os_collection`
`contains`	`( const E ) const`	`os_int32`	`os_Collection`
`count`	`( const E ) const`	`os_int32`	`os_Collection`
`empty`	`( )`	`os_int32`	`os_collection`
`get_behavior`	`( ) const`	`os_unsigned_int32`	`os_collection`
`insert`	`( const E )`	`void`	`os_Collection`
`insert_after`	`( const void*, const os_cursor& )` `( const void*, os_unsigned_int32 )`	`void`	`os_collection`

Name	Arguments	Returns	Defined By
insert_before	(const void*, const os_cursor&) (const void*, os_unsigned_int32)	void	os_collection
insert_first	(const void*)	void	os_collection
insert_last	(const void*)	void	os_collection
only	( ) const	E	os_Collection
operator os_Array<E>&	( )		os_Collection
operator const os_Array<E>&	( ) const		os_Collection
operator os_array&	( )		os_collection
operator const os_array&	( ) const		os_collection
operator os_bag&	( )		os_collection
operator const os_bag&	( ) const		os_collection
operator os_List<E>&	( )		os_Collection
operator const os_List<E>&	( ) const		os_Collection
operator os_list&	( )		os_collection
operator const os_list&	( ) const		os_collection
operator os_Set<E>&	( )		os_Collection
operator const os_Set<E>&	( ) const		os_Collection
operator os_set&	( )		os_collection
operator const os_set&	( ) const		os_collection
operator ==	(const os_Collection<E>&) const (const E) const	os_int32 os_int32	os_Collection
operator !=	(const os_Collection<E>&) const (const E) const	os_int32 os_int32	os_Collection
operator <	(const os_Collection<E>&) const (const E) const	os_int32 os_int32	os_Collection
operator <=	(const os_Collection<E>&) const (const E) const	os_int32 os_int32	os_Collection
operator >	(const os_Collection<E>&) const (const E) const	os_int32 os_int32	os_Collection

<table>
<tr><th>Name</th><th>Arguments</th><th>Returns</th><th>Defined By</th></tr>
<tr><td>operator >=</td><td>(const os_Collection<E>&) const
(const E) const</td><td>os_int32
os_int32</td><td>os_Collection</td></tr>
<tr><td>operator =</td><td>(const os_Bag<E>&) const
(const os_Collection<E>&) const
(const E) const</td><td>os_Bag<E>&
os_Bag<E>&
os_Bag<E>&</td><td>os_Bag</td></tr>
<tr><td>operator |=</td><td>(const os_Collection<E>&) const
(const E) const</td><td>os_Bag<E>&
os_Bag<E>&</td><td>os_Bag</td></tr>
<tr><td>operator |</td><td>(const os_Collection<E>&) const
(const E) const</td><td>os_Collection<E>&
os_Collection<E>&</td><td>os_Collection</td></tr>
<tr><td>operator &=</td><td>(const os_Collection<E>&) const
(const E) const</td><td>os_Bag<E>&
os_Bag<E>&</td><td>os_Bag</td></tr>
<tr><td>operator &</td><td>(const os_Collection<E>&) const
(const E) const</td><td>os_Collection<E>&
os_Collection<E>&</td><td>os_Collection</td></tr>
<tr><td>operator -=</td><td>(const os_Collection<E>&) const
(const E) const</td><td>os_Bag<E>&
os_Bag<E>&</td><td>os_Bag</td></tr>
<tr><td>operator -</td><td>(const os_Collection<E>&) const
(const E) const</td><td>os_Collection<E>&
os_Collection<E>&</td><td>os_Collection</td></tr>
<tr><td>os_Bag</td><td>( )
(os_int32 expected_size)
(const os_Bag<E>&)
(const os_Collection<E>&)</td><td></td><td>os_Bag</td></tr>
<tr><td>remove</td><td>(const E)</td><td>os_int32</td><td>os_Collection</td></tr>
<tr><td>remove_at</td><td>(const os_Cursor<E>&)</td><td>void</td><td>os_Collection</td></tr>
<tr><td>remove_first</td><td>(const void*&)
( )</td><td>os_int32
void*</td><td>os_collection</td></tr>
<tr><td>remove_last</td><td>(const void*&)
( )</td><td>os_int32
void*</td><td>os_collection</td></tr>
<tr><td>replace_at</td><td>(const E,
const os_Cursor<E>&)</td><td>E</td><td>os_Collection</td></tr>
<tr><td>retrieve</td><td>(const os_Cursor<E>&) const</td><td>E</td><td>os_Collection</td></tr>
</table>

Name	*Arguments*	*Returns*	*Defined By*
`retrieve_first`	`( ) const` `( const void*& ) const`	`void*` `os_int32`	`os_collection`
`retrieve_last`	`( ) const` `( const void*& ) const`	`void*` `os_int32`	`os_collection`

os_Bag enumerators

The following table lists the enumerators inherited by `os_Bag` from `os_collection`.

Enumerator	*Inherited From*
`allow_nulls`	`os_collection`
`EQ`	`os_collection`
`GT`	`os_collection`
`LT`	`os_collection`
`maintain_order`	`os_collection`

Assignment Operator Semantics

Assignment operator semantics are described for the following functions in terms of insert operations into the target collection. The actual implementation of the assignment might be different, but the associated semantics are maintained.

os_Bag::operator =()

```
os_Bag<E> &operator =(const os_Collection<E> &s);
```

```
os_Bag<> &operator=(const os_Bag<E> &s);
```

Copies the contents of the collection *s* into the target collection and returns the target collection. The copy is performed by effectively clearing the target, iterating over the source collection, and inserting each element into the target collection. The target collection semantics are enforced as usual during the insertion process.

```
os_Bag<E> &operator =(const E e);
```

Clears the target collection, inserts the element *e* into the target collection, and returns the target collection.

os_Bag::operator |=()

```
os_Bag<E> &operator |=(const os_Collection<E> &s);
```

Inserts the elements contained in *s* into the target collection and returns the target collection.

```
os_Bag<E> &operator |=(const E e);
```

Inserts the element *e* into the target collection and returns the target collection.

os_Bag::operator &=()

```
os_Bag<E> &operator &=(const os_Collection<E> &s);
```

For each element in the target collection, reduces the count of the element in the target to the minimum of the counts in the source and target collections. The function returns the target collection.

```
os_Bag<E> &operator &=(const E e);
```

If *e* is present in the target, converts the target into a collection containing just the element *e*. Otherwise, the function clears the target collection. The function returns the target collection.

os_Bag::operator –=()

```
os_Bag<E> &operator –=(const os_Collection<E> &s);
```

For each element in the collection *s*, removes `s.count(e)` occurrences of the element from the target collection. It returns the target collection.

```
os_Bag<E> &operator –=(const E e);
```

Removes the element *e* from the target collection. The function returns the target collection.

os_Bag::os_Bag()

```
os_Bag();
```

Returns an empty bag.

```
os_Bag(os_int32);
```

Returns an empty bag whose initial implementation is based on the expectation that the specified `os_int32` indicates the approximate usual size of the bag, after it has been loaded with elements.

```
os_Bag(const os_Bag<E>&);
```

Returns a bag that results from assigning the specified bag to an empty bag.

```
os_Bag(const os_Collection<E>&);
```

Returns a bag that results from assigning the specified collection to an empty bag.

os_bag

`class os_bag : public os_collection`

A bag (sometimes called a *multiset*) is an unordered collection. Unlike values in sets, values can occur in a bag more than once at a given time. The *count* of a value in a given bag is the number of times it occurs in the bag. Repeated insertion of a value into a bag increases its count in the bag by 1 each time. The count of a value in a bag is 0 if and only if the value is not an element of the bag.

The class `os_bag` is nonparameterized. For the parameterized version of this class, see `os_Bag` on page 73.

Tables of member functions and enumerators

The first of the following tables lists the member functions that can be performed on instances of `os_bag`. The second table lists the enumerators inherited by `os_bag` from `os_collection`. Many functions are also inherited by `os_bag` from `os_collection`. The full explanation of each inherited function or enumerator appears in the entry for the class from which it is inherited. The full explanation of each function defined by `os_bag` appears in this entry, after the tables. In each case, the *Defined By* column gives the class whose entry contains the full explanation.

Name	*Arguments*	*Returns*	*Defined By*
`cardinality`	`( ) const`	`os_int32`	`os_collection`
`clear`	`( )`	`void`	`os_collection`
`contains`	`( const void* ) const`	`os_int32`	`os_collection`
`count`	`( const void* ) const`	`os_int32`	`os_collection`
`empty`	`( )`	`os_int32`	`os_collection`
`get_behavior`	`( ) const`	`os_unsigned_ int32`	`os_collection`
`insert`	`( const void* )`	`void`	`os_collection`
`only`	`( ) const`	`void*`	`os_Collection`
`operator os_array&`	`( )`		`os_collection`
`operator const os_array&`	`( ) const`		`os_collection`
`operator os_list&`	`( )`		`os_collection`
`operator const os_list&`	`( ) const`		`os_collection`
`operator os_set&`	`( )`		`os_collection`
`operator const os_set&`	`( ) const`		`os_collection`
`operator ==`	`( const os_collection& ) const` `( const void* ) const`	`os_int32` `os_int32`	`os_collection`
`operator !=`	`( const os_collection& ) const` `( const void* ) const`	`os_int32` `os_int32`	`os_collection`

<table>
<tr><th>Name</th><th>Arguments</th><th>Returns</th><th>Defined By</th></tr>
<tr><td>operator <</td><td>(const os_collection&) const
(const void*) const</td><td>os_int32
os_int32</td><td>os_collection</td></tr>
<tr><td>operator <=</td><td>(const os_collection&) const
(const void*) const</td><td>os_int32
os_int32</td><td>os_collection</td></tr>
<tr><td>operator ></td><td>(const os_collection&) const
(const void*) const</td><td>os_int32
os_int32</td><td>os_collection</td></tr>
<tr><td>operator >=</td><td>(const os_collection&) const
(const void*) const</td><td>os_int32
os_int32</td><td>os_collection</td></tr>
<tr><td>operator =</td><td>(const os_bag&) const
(const os_collection&) const
(const void*) const</td><td>os_bag&
os_bag&
os_bag</td><td>os_bag</td></tr>
<tr><td>operator |=</td><td>(const os_collection&) const
(const void*) const</td><td>os_bag&
os_bag&</td><td>os_bag</td></tr>
<tr><td>operator |</td><td>(const os_collection&) const
(const void*) const</td><td>os_collection&
os_collection&</td><td>os_collection</td></tr>
<tr><td>operator &=</td><td>(const os_collection&) const
(const void*) const</td><td>os_bag&
os_bag&</td><td>os_bag</td></tr>
<tr><td>operator &</td><td>(const os_collection&) const
(const void*) const</td><td>os_collection&
os_collection&</td><td>os_collection</td></tr>
<tr><td>operator -=</td><td>(const os_collection&) const
(const void*) const</td><td>os_bag&
os_bag&</td><td>os_bag</td></tr>
<tr><td>operator -</td><td>(const os_collection&) const
(const void*) const</td><td>os_collection&
os_collection&</td><td>os_collection</td></tr>
<tr><td>os_bag</td><td>( )
(os_int32 expected_size)
(const os_bag&)
(const os_collection&)</td><td></td><td>os_bag</td></tr>
<tr><td>remove</td><td>(const void*)</td><td>os_int32</td><td>os_collection</td></tr>
<tr><td>remove_at</td><td>(const os_cursor&)</td><td>void</td><td>os_collection</td></tr>
<tr><td>replace_at</td><td>(const void*,
const os_cursor&)</td><td>void*</td><td>os_collection</td></tr>
<tr><td>retrieve</td><td>(const os_cursor&) const</td><td>void*</td><td>os_collection</td></tr>
</table>

os_bag enumerators

The following table lists the enumerators inherited by `os_bag` from `os_collection`.

Name	*Inherited From*
allow_nulls	os_collection
EQ	os_collection
GT	os_collection
LT	os_collection
maintain_order	os_collection

Assignment Operator Semantics

Assignment operator semantics are described for the following functions in terms of insert operations into the target collection. The actual implementation of the assignment might be different, but the associated semantics are maintained.

os_bag::operator =()

```
os_bag &operator = (const os_collection &s);
```

Copies the contents of the collection *s* into the target collection and returns the target collection. The copy is performed by effectively clearing the target, iterating over the source collection, and inserting each element into the target collection. The target collection semantics are enforced as usual during the insertion process.

```
os_bag &operator =(const void *e);
```

Clears the target collection, inserts the element *e* into the target collection, and returns the target collection.

os_bag::operator |=()

```
os_bag &operator |=(const os_bag &s);
```

Inserts the elements contained in *s* into the target collection and returns the target collection.

```
os_bag &operator |=(const void *e);
```

Inserts the element *e* into the target collection and returns the target collection.

os_bag::operator &=()

```
os_bag &operator &=(const os_collection &s);
```

For each element in the target collection, reduces the count of the element in the target to the minimum of the counts in the source and target collections. The function returns the target collection.

```
os_bag &operator &=(const void *e);
```

If *e* is present in the target, converts the target into a collection containing just the element *e*. Otherwise, it clears the target collection. The function returns the target collection.

os_bag::operator –=()

```
os_bag &operator –=(const os_collection &s);
```

For each element in the collection *s*, removes `s.count(e)` occurrences of the element from the target collection. The function returns the target collection.

```
os_bag &operator –=(const void *e);
```

Removes the element *e* from the target collection. It returns the target collection.

os_bag::os_bag()

```
os_bag();
```

Returns an empty bag.

```
os_bag(os_int32);
```

Returns an empty bag whose initial implementation is based on the expectation that the specified `os_int32` indicates the approximate usual cardinality of the bag, after it has been loaded with elements.

```
os_bag(const os_bag&);
```

Returns a bag that results from assigning the specified bag to an empty bag.

```
os_bag(const os_collection&);
```

Returns a bag that results from assigning the specified collection to an empty bag.

os_Collection

```
template <class E>
class os_Collection : public os_collection
```

The `os_Collection` class is an interface class. It defines functions that operate on collections. Use it as the declaration type for arguments to user-defined functions that take the collections subtype `os_Array`, `os_Bag`, `os_Set`, or `os_List`. Also, use it as an abstract base class for deriving other collections subtypes. Do not instantiate `os_Collection`.

A collection is an object that serves to group together other objects. The objects so grouped are the collection's *elements*. For some collections, a value can occur as an element more than once. The *count* of a value in a given collection is the number of times (possibly 0) it occurs in the collection.

The class `os_Collection` is *parameterized*, with a parameter for constraining the type of values allowable as elements (for the nonparameterized version of this class, see `os_collection` on page 95). This means that when specifying `os_Collection` as a function's formal parameter or as the type of a variable or data member, you must specify the the collection's *element type* parameter. This is accomplished by appending to `os_Collection` the name of the element type enclosed in angle brackets (< >):

`os_Collection<`*`element-type-name`*`>`

The element type parameter, `E`, occurs in the signatures of some of the functions described below. The parameter is used by the compiler to detect type errors.

The element type of any instance of `os_Collection` must be a pointer type.

Tables of member functions and enumerators

The first of the following tables lists the member functions that can be performed on instances of `os_Collection`. The second table lists the enumerators inherited by `os_Collection` from `os_collection`. Many functions are also inherited by `os_Collection` from `os_collection`. The full explanation of each inherited function or enumerator appears in the entry for the class from which it is inherited. The full explanation of each function defined by `os_Collection` appears in this entry, after the tables. In each case, the *Defined By* column gives the class whose entry contains the full explanation.

Name	*Arguments*	*Returns*	*Defined By*
`cardinality`	`( ) const`	`os_unsigned_ int32`	`os_collection`
`clear`	`( )`	`void`	`os_collection`
`contains`	`( const E ) const`	`os_int32`	`os_Collection`
`count`	`( const E ) const`	`os_int32`	`os_Collection`
`empty`	`( )`	`os_int32`	`os_collection`
`get_behavior`	`( ) const`	`os_unsigned_ int32`	`os_collection`
`insert`	`( const E )`	`void`	`os_Collection`

Name	*Arguments*	*Returns*	*Defined By*
insert_after	(const E, const os_Cursor<E>&) (const E, os_unsigned_int32)	void void	os_Collection
insert_before	(const E, const os_Cursor<E>&) (const E, os_unsigned_int32)	void void	os_Collection
insert_first	(const E)	void	os_Collection
insert_last	(const E)	void	os_Collection
only	( ) const	E	os_Collection
operator os_Array<E>&	( )		os_Collection
operator const os_Array<E>&	( ) const		os_Collection
operator os_array&	( )		os_collection
operator const os_array&	( ) const		os_collection
operator os_Bag<E>&	( )		os_Collection
operator const os_Bag<E>&	( ) const		os_Collection
operator os_bag&	( )		os_collection
operator const os_bag&	( ) const		os_collection
operator os_List<E>&	( )		os_Collection
operator const os_List<E>&	( ) const		os_Collection
operator os_list&	( )		os_collection
operator const os_list&	( ) const		os_collection
operator os_Set<E>&	( )		os_Collection
operator const os_Set<E>&	( ) const		os_Collection
operator os_set&	( )		os_collection
operator const os_set&	( ) const		os_collection
operator ==	(const os_Collection<E>&) const (E) const	os_int32 os_int32	os_Collection

Name	*Arguments*	*Returns*	*Defined By*
operator !=	(const os_Collection<E>&) const (E) const	os_int32 os_int32	os_Collection
operator <	(const os_Collection<E>&) const (E) const	os_int32 os_int32	os_Collection
operator <=	(const os_Collection<E>&) const (E) const	os_int32 os_int32	os_Collection
operator >	(const os_Collection<E>&) const (E) const	os_int32 os_int32	os_Collection
operator >=	(const os_Collection<E>&) const (E) const	os_int32 os_int32	os_Collection
operator =	(const os_Collection<E>&) const (E) const	os_Collection<E>& os_Collection<E>&	os_Collection
operator |=	(const os_Collection<E>&) const (E) const	os_Collection<E>& os_Collection<E>&	os_Collection
operator |	(const os_Collection<E>&) const (E) const	os_Collection<E>& os_Collection<E>&	os_Collection
operator &=	(const os_Collection<E>&) const (E) const	os_Collection<E>& os_Collection<E>&	os_Collection
operator &	(const os_Collection<E>&) const (E) const	os_Collection<E>& os_Collection<E>&	os_Collection
operator –=	(const os_Collection<E>&) const (E) const	os_Collection<E>& os_Collection<E>&	os_Collection
operator -	(const os_Collection<E>&) const (E) const	os_Collection<E>& os_Collection<E>&	os_Collection
remove	(const E)	os_int32	os_Collection
remove_at	(const os_Cursor<E>&) (os_unsigned_int32)	void void	os_Collection

Name	*Arguments*	*Returns*	*Defined By*
remove_first	(const E&) ( )	os_int32 E	os_Collection
remove_last	(const E&) ( )	os_int32 E	os_Collection
replace_at	(const E, const os_Cursor<E>&) (E, os_unsigned_int32)	E E	os_Collection
retrieve	(os_unsigned_int32) const (const os_Cursor<E>&) const	E E	os_Collection
retrieve_first	( ) const (E&) const	E os_int32	os_Collection
retrieve_last	( ) const (E&) const	E os_int32	os_Collection

os_Collection enumerators

The following table lists the enumerators for `os_Collection`.

Name	*Inherited From*
allow_duplicates	os_collection
allow_nulls	os_collection
EQ	os_collection
GT	os_collection
LT	os_collection
maintain_order	os_collection

os_Collection::contains()

```
os_int32 contains(E) const;
```

Returns a nonzero `os_int32` if the specified `E` is an element of the specified collection, and `0` otherwise.

os_Collection::count()

```
os_int32 count(E);
```

Returns the number of occurrences (possibly 0) of the specified `E` in the collection for which the function was called.

os_Collection::insert()

```
void insert(const E);
```

Adds the specified instance of `E` to the collection for which the function was called. The behavior of `insert()` depends on the characteristics of the collection you are using:

- If the collection is ordered, the element is inserted at the end of the collection.
- If the collection disallows nulls and the specified `E` is `0`, `err_coll_nulls` is signaled.
- If the collection disallows duplicates and the specified `E` is already an element of the collection, the insertion is ignored; that is, nothing is inserted into the collection.

os_Collection::insert_after()

```
void insert_after(const E, const os_Cursor<E>&);
```

Adds the specified instance of `E` to the collection for which the function was called. The new element is inserted immediately after the element at which the specified cursor is positioned. The index of all elements after the new element increases by 1. The cursor must be a default cursor (that is, one that results from a constructor call with only a single argument). If the cursor is null, `err_coll_null_cursor` is signaled. If the cursor is invalid, `err_coll_illegal_cursor` is signaled.

```
void insert_after(const E, os_unsigned_int32);
```

Adds the specified instance of `E` to the collection for which the function was called. The new element is inserted after the position indicated by the `os_unsigned_int32`. The index of all elements after the new element increases by 1. If the index is not less than the collection's cardinality, `err_coll_out_of_range` is signaled.

The behavior of `insert_after()` (both signatures) depends on the characteristics of the collection you are using:

- If the collection is not ordered, `err_coll_not_supported` is signaled.
- If the collection disallows nulls and the specified `E` is `0`, `err_coll_nulls` is signaled.
- If the collection is an array, all elements after this element are pushed down.
- If the collection disallows duplicates and the specified `E` is already an element of the collection, the insertion is ignored; that is, nothing is inserted into the collection.

os_Collection::insert_before()

```
void insert_before(const E, const os_Cursor<E>&);
```

Adds the specified instance of `E` to the collection for which the function was called. The new element is inserted immediately before the element at which the specified cursor is positioned. The index of all elements after the new element increases by 1. The cursor must be a default cursor (that is, one that results from a constructor call with only a single argument). If the cursor is null, `err_coll_null_cursor` is signaled. If the cursor is invalid, `err_coll_illegal_cursor` is signaled.

```
void insert_before(const E, os_unsigned_int32);
```

Adds the specified instance of `E` to the collection for which the function was called. The new element is inserted immediately before the position indicated by the `os_unsigned_int32`. The index of all elements after the new element increases by 1. If the index is not less than the collection's cardinality, `err_coll_out_of_range` is signaled.

The behavior of `insert_before()` (both signatures) depends on the characteristics of the collection you are using:

- If the collection is not ordered, `err_coll_not_supported` is signaled.
- If the collection disallows nulls and the specified `E` is `0`, `err_coll_nulls` is signaled.
- If the collection is an array, all elements after the inserted element are pushed down.
- If the collection disallows duplicates and the specified `E` is already an element of the collection, the insertion is ignored; that is, nothing is inserted into the collection.

os_Collection::insert_first()

```
void insert_first(const E);
```

Adds the specified instance of `E` to the beginning of the collection for which the function was called. The behavior of `insert_first()` depends on the characteristics of the collection you are using:

- If the collection is not ordered, `err_coll_not_supported` is signaled.
- If the collection disallows nulls and the specified `E` is `0`, `err_coll_nulls` is signaled.
- If the collection is an array, all elements after the inserted element are pushed down.
- If the collection disallows duplicates and the specified `E` is already an element of the collection, the insertion is ignored; that is, nothing is inserted into the collection.

os_Collection::insert_last()

```
void insert_last(const E);
```

Adds the specified instance of `E` to the end of the collection for which the function was called.

- If the collection is not ordered, `err_coll_not_supported` is signaled.
- If the collection disallows nulls and the specified `E` is `0`, `err_coll_nulls` is signaled.
- If the collection disallows duplicates and the specified `E` is already an element of the collection, the insertion is ignored; that is, nothing is inserted into the collection.

os_Collection::only()

```
E only() const;
```

Returns the only element of the specified collection. If the collection has more than one element, `err_coll_not_singleton` is signaled. If the collection is empty, `0` is returned.

os_Collection::operator os_Array()

```
operator os_Array<E>&();
```

Returns an array with the same elements and behavior as the specified collection. An exception is signaled if the collection's behavior is incompatible with the required behavior of arrays.

os_Collection::operator const os_Array()

```
operator const os_Array<E>&() const;
```

Returns an array with the same elements and behavior as the specified collection. An exception is signaled if the collection's behavior is incompatible with the required behavior of arrays.

os_Collection::operator os_Bag()

```
operator os_Bag<E>&();
```

Returns a bag with the same elements and behavior as the specified collection. An exception is signaled if the collection's behavior is incompatible with the required behavior of bags.

os_Collection::operator const os_Bag()

```
operator const os_Bag<E>&() const;
```

Returns a bag with the same elements and behavior as the specified collection. An exception is signaled if the collection's behavior is incompatible with the required behavior of bags.

os_Collection::operator os_List()

```
operator os_List<E>&();
```

Returns a list with the same elements and behavior as the specified collection. An exception is signaled if the collection's behavior is incompatible with the required behavior of lists.

os_Collection::operator const os_List()

```
operator const os_List<E>&() const;
```

Returns a list with the same elements and behavior as the specified collection. An exception is signaled if the collection's behavior is incompatible with the required behavior of lists.

os_Collection::operator os_Set()

```
operator os_Set<E>&();
```

Returns a set with the same elements and behavior as the specified collection. An exception is signaled if the collection's behavior is incompatible with the required behavior of sets.

os_Collection::operator const os_Set()

```
operator const os_Set<E>&() const;
```

Returns a set with the same elements and behavior as the specified collection. An exception is signaled if the collection's behavior is incompatible with the required behavior of sets.

os_Collection::operator ==()

```
os_int32 operator ==(const os_Collection<E> &s) const;
```

Returns a nonzero value if and only if for each element in the `this` collection `count(`*`element`*`) == `*`s`*`.count(`*`element`*`)` and both collections have the same cardinality. Note that the comparison does not take order into account.

```
os_int32 operator ==(const E s) const;
```

Returns a nonzero value if and only if the collection contains *s* and nothing else.

os_Collection::operator !=()

```
os_int32 operator !=(const os_Collection<E> &s) const;
```

Returns a nonzero value if and only if it is not the case both that (1) for each element in the `this` collection, `count(element) == `*`s`*`.count(element)`, and (2) both collections have the same cardinality. Note that the comparison does not take order into account.

```
os_int32 operator !=(const E s) const;
```

Returns a nonzero value if and only if it is not the case that the collection contains *s* and nothing else.

os_Collection::operator <()

```
os_int32 operator <(const os_Collection<E> &s) const;
```

Returns a nonzero value if and only if for each element in the this collection count(element) <= *s*.count(element) and cardinality() < *s*.cardinality().

```
os_int32 operator <(const E s) const;
```

Returns a nonzero value if and only if the specified collection is empty.

os_Collection::operator <=()

```
os_int32 operator <=(const os_Collection<E> &s) const;
```

Returns a nonzero value if and only if for each element in the this collection count(element) <= *s*.count(element).

```
os_int32 operator <=(const E s) const;
```

Returns a nonzero value if and only if the specified collection is empty or *e* is the only element in the collection.

os_Collection::operator >()

```
os_int32 operator >(const os_Collection<E> &s) const;
```

Returns a nonzero value if and only if for each element of *s*.count(element) >= *s*.count(element) and cardinality() > *s*.cardinality().

```
os_int32 operator >(const E s) const;
```

Returns a nonzero value if and only if count(*s*) >= 1 and cardinality() > 1.

os_Collection::operator >=()

```
os_int32 operator >=(const os_Collection<E> &s) const;
```

Returns a nonzero value if and only if for each element of *s* count(element) >= *s*.count(element).

```
os_int32 operator >=(const E s) const;
```

Returns a nonzero value if and only if count(*s*) >= 1.

Assignment Operator Semantics

Assignment operator semantics are described for the following functions in terms of insert operations into the target collection. The actual implementation of the assignment might be different, but the associated semantics are maintained.

os_Collection::operator =()

```
os_Collection<E> &operator =(const os_Collection<E> &s);
```

Copies the contents of the collection *s* into the target collection and returns the target collection. The copy is performed by effectively clearing the target, iterating over the source collection, and inserting each element into the target collection. The iteration is ordered if the source collection is ordered. The target collection semantics are enforced as usual during the insertion process.

```
os_Collection<E> &operator =(const E e);
```

Clears the target collection, inserts the element *e* into the target collection, and returns the target collection.

os_Collection::operator |=()

```
os_Collection<E> &operator |=(const os_Collection<E> &s);
```

Inserts the elements contained in *s* into the target collection and returns the target collection.

```
os_Collection<E> &operator |=(const E e);
```

Inserts the element *e* into the target collection and returns the target collection.

os_Collection::operator |()

```
os_Collection<E> &operator |(const os_Collection<E> &s) const;
```

Copies the contents of `this` into a new collection, `c`, and then performs `c |= s`. The new collection, `c`, is then returned. If either operand allows duplicates or nulls, the result does. If both operands maintain order, the result does.

```
os_Collection<E> &operator |(const E e) const;
```

Copies the contents of `this` into a new collection, `c`, and then performs `c |= e`. The new collection, `c`, is then returned. If `this` allows duplicates, maintains order, or allows nulls, the result does.

os_Collection::operator &=()

```
os_Collection<E> &operator &=(const os_Collection<E> &s);
```

For each element in the target collection, reduces the count of the element in the target to the minimum of the counts in the source and target collections. If the collection is ordered and contains duplicates, it does so by retaining the appropriate number of leading elements. The function returns the target collection.

```
os_Collection<E> &operator &=(const E e);
```

If *e* is present in the target, converts the target into a collection containing just the element *e*. Otherwise, it clears the target collection. The function returns the target collection.

os_Collection::operator &()

```
os_Collection<E> &operator &(const os_Collection<E> &s) const;
```

Copies the contents of `this` into a new collection, `c`, and then performs `c &= s`. The new collection, `c`, is then returned. If either operand allows duplicates or nulls, the result does. If both operands maintain order, the result does.

```
os_Collection<E> &operator &(E e) const;
```

Copies the contents of `this` into a new collection, `c`, and then performs `c &= e`. The new collection, `c`, is then returned. If `this` allows duplicates, maintains order, or allows nulls, the result does.

os_Collection::operator –=()

```
os_Collection<E> &operator –=(const os_Collection<E> &s);
```

For each element in the collection *s*, removes `s.count(e)` occurrences of the element from the target collection. If the collection is ordered, it is the first `s.count(e)` elements that are removed. It returns the target collection.

```
os_Collection<E> &operator –=(E e);
```

Removes the element *e* from the target collection. If the collection is ordered, it is the first occurrence of the element that is removed from the target collection. It returns the target collection.

os_Collection::operator –()

```
os_Collection<E> &operator –(const os_Collection<E> &s) const;
```

Copies the contents of `this` into a new collection, `c`, and then performs `c –= s`. The new collection, `c`, is then returned. If either operand allows duplicates or nulls, the result does. If both operands maintain order, the result does.

```
os_Collection<E> &operator –(E e) const;
```

Copies the contents of `this` into a new collection, `c`, and then performs `c –= s`. The new collection, `c`, is then returned. If `this` allows duplicates, maintains order, or allows nulls, the result does.

os_Collection::remove()

```
os_int32 remove(const E);
```

Removes the specified instance of `E` from the collection for which the function was called, if present. If the collection is ordered, the first occurrence of the specified `E` is removed. If the collection is an array, all elements after this element are pushed up.

os_Collection::remove_first()

```
os_int32 remove_first(const E&);
```

Removes the first element from the specified collection and returns the removed element, or `0` if the collection is empty. If successful, the function modifies its argument to refer to the removed element. If the specified collection is not ordered, `err_coll_not_supported` is signaled. If the collection is an array, all elements after the removed element are pushed up.

```
E remove_first();
```

Removes the first element from the specified collection and returns the removed element, or `0` if the collection is empty. Note that for collections that allow null elements, the significance of the return value can be ambiguous. The preceding alternative overloading of `remove_first()` can be used to avoid the ambiguity. If the specified collection is not ordered, `err_coll_not_supported` is signaled. If the collection is an array, all elements after the removed element are pushed up.

os_Collection::remove_last()

```
os_int32 remove_last(const E&);
```

Removes the last element from the specified collection if the collection is not empty, returns a nonzero `os_int32` if the collection is not empty, and modifies its argument to refer to the removed element. If the specified collection is not ordered, `err_coll_not_supported` is signaled.

```
E remove_last();
```

Removes the last element from the specified collection and returns the removed element, or `0` if the collection was empty. Note that for collections that allow null elements, the significance of the return value can be ambiguous. The preceding alternative overloading of `remove_last()` can be used to avoid the ambiguity. If the specified collection is not ordered, `err_coll_not_supported` is signaled.

os_Collection::replace_at()

```
E replace_at(const E, const os_Cursor<E>&);
```

Returns the element at which the specified cursor is positioned and replaces it with the specified instance of `E`. The cursor must be a default cursor (that is, one that results from a constructor call with only a single argument). If the cursor is null, `err_coll_null_cursor` is signaled. If the cursor is invalid, `err_coll_illegal_cursor` is signaled.

```
E replace_at(const E, os_unsigned_int32 position);
```

Returns the element at the specified position and replaces it with the specified instance of `E`. If the position is not less than the collection's cardinality, `err_coll_out_of_range` is signaled. If the collection is not ordered, `err_coll_not_supported` is signaled.

os_Collection::retrieve()

```
E retrieve(const os_Cursor<E>&) const;
```

Returns the element at which the specified cursor is positioned. The cursor must be a default cursor (that is, one that results from a constructor call with only a single argument). If the cursor is null, `err_coll_null_cursor` is signaled. If the cursor is invalid, `err_coll_illegal_cursor` is signaled.

```
E retrieve(os_unsigned_int32 position) const;
```

Returns the element at the specified position. If the position is not less than the collection's cardinality, `err_coll_out_of_range` is signaled. If the collection is not ordered, `err_coll_not_supported` is signaled.

os_Collection::retrieve_first()

```
E retrieve_first() const;
```

Returns the specified collection's first element, or `0` if the collection is empty. For collections that contain zeros, see the other overloading of this function, following. If the collection is not ordered, `err_coll_not_supported` is signaled.

```
os_int32 retrieve_first(const E&) const;
```

Returns `0` if the specified collection is empty; returns a nonzero `os_int32` otherwise. Modifies the argument to refer to the collection's first element. If the collection is not ordered, `err_coll_not_supported` is signaled.

os_Collection::retrieve_last()

```
E retrieve_last() const;
```

Returns the specified collection's last element, or `0` if the collection is empty. For collections that contain zeros, see the other overloading of this function, following. If the collection is not ordered, `err_coll_not_supported` is signaled.

```
os_int32 retrieve_last(const E&) const;
```

Returns `0` if the specified collection is empty; returns a nonzero `os_int32` otherwise. Modifies the argument to refer to the collection's last element. If the collection is not ordered, `err_coll_not_supported` is signaled.

os_collection

A collection is an object that serves to group together other objects. The objects so grouped are the collection's *elements*. For some collections, a value can occur as an element more than once. The *count* of a value in a given collection is the number of times (possibly 0) it occurs in the collection.

Like an `os_Collection`, an `os_collection` is not meant to be instantiated. It is a base class for the other collection subtypes. It can be used as a generic handle to be passed around to user-defined functions that can take any collection subtype as an argument.

This class has a parameterized subtype. See `os_Collection` on page 82.

The element type of any instance of `os_collection` must be a pointer type.

In addition, the static member function `os_collection::initialize()` must be executed in a process before using any ObjectStore collection or relationship functionality is made.

Tables of member functions and enumerators

The first of the following tables lists the member functions defined by `os_collection`, together with their formal argument lists and return types. The second table lists the enumerators defined by `os_collection`. The full explanation of each function and enumerator follows these tables.

Name	*Arguments*	*Returns*
`cardinality`	`( ) const`	`os_int32`
`clear`	`( )`	`void`
`contains`	`( const void* ) const`	
`count`	`( const void* ) const`	`os_int32`
`empty`	`( )`	`os_int32`
`get_behavior`	`( ) const`	`os_unsigned_int32`
`insert`	`( const void* )`	`void`

`insert_after`	`( const void*, const os_cursor& )`
	
	`( const void*, os_unsigned_int32 )`
		`void`
		
		`void`

`insert_before`	`( const void*, const os_cursor& )`
	
	`( const void*, os_unsigned_int32 )`
		`void`
		
		`void`

`insert_first`	`( const void* )`	`void`
`insert_last`	`( const void* )`	`void`
`only`	`( ) const`	`void*`
`operator os_array&`	`( )`	
`operator const os_array&`	`( ) const`	

<table>
<tr><th>Name</th><th>Arguments</th><th>Returns</th></tr>
<tr><td>operator os_bag&</td><td>( )</td><td></td></tr>
<tr><td>operator const os_bag&</td><td>( ) const</td><td></td></tr>
<tr><td>operator os_list&</td><td>( )</td><td></td></tr>
<tr><td>operator const os_list&</td><td>( ) const</td><td></td></tr>
<tr><td>operator os_set&</td><td>( )</td><td></td></tr>
<tr><td>operator const os_set&</td><td>( ) const</td><td></td></tr>
<tr><td>operator ==</td><td>(const os_collection&) const
(const void*) const</td><td>os_int32
os_int32</td></tr>
<tr><td>operator !=</td><td>(const os_collection&) const
(const void*) const</td><td>os_int32
os_int32</td></tr>
<tr><td>operator <</td><td>(const os_collection&) const
(const void*) const</td><td>os_int32
os_int32</td></tr>
<tr><td>operator <=</td><td>(const os_collection&) const
(const void*) const</td><td>os_int32
os_int32</td></tr>
<tr><td>operator ></td><td>(const os_collection&) const
(const void*) const</td><td>os_int32
os_int32</td></tr>
<tr><td>operator >=</td><td>(const os_collection&) const
(const void*) const</td><td>os_int32
os_int32</td></tr>
<tr><td>operator =</td><td>(const os_collection&) const
(const void*) const</td><td>os_collection&
os_collection&</td></tr>
<tr><td>operator |=</td><td>(const os_collection&) const
(const void*) const</td><td>os_collection&
os_collection&</td></tr>
<tr><td>operator |</td><td>(const os_collection&) const
(const void*) const</td><td>os_collection&
os_collection&</td></tr>
<tr><td>operator &=</td><td>(const os_collection&) const
(const void*) const</td><td>os_collection&
os_collection&</td></tr>
<tr><td>operator &</td><td>(const os_collection&) const
(const void*) const</td><td>os_collection&
os_collection&</td></tr>
<tr><td>operator –=</td><td>(const os_collection&) const
(const void*) const</td><td>os_collection&
os_collection&</td></tr>
<tr><td>operator -</td><td>(const os_collection&) const
(const void*) const</td><td>os_collection&
os_collection&</td></tr>
<tr><td>remove</td><td>(const void*)</td><td>os_int32</td></tr>
<tr><td>remove_at</td><td>(const os_cursor&)
(os_unsigned_int32)</td><td>void
void</td></tr>
<tr><td>remove_first</td><td>(const void*&)
( )</td><td>os_int32
void*</td></tr>
</table>

Name	*Arguments*	*Returns*
remove_last	(const void*&) ( )	os_int32 void*
replace_at	(const void*, const os_cursor&) (const void*, os_unsigned_int32)	void* void*
retrieve	(os_unsigned_int32) const (const os_cursor&) const	void* void*
retrieve_first	( ) const (const void*&) const	void* os_int32
retrieve_last	( ) const (const void*&) const	void* os_int32

os_collection enumerators

The following lists enumerators for `os_collection`.

Enumerators
- `allow_duplicates`
- `allow_nulls`
- `EQ`
- `GE`
- `GT`
- `LE`
- `LT`
- `maintain_order`
- `NE`
- `optimized_list`

os_collection::allow_duplicates

Possible element of the bit-wise disjunction return value of the `os_collection::get_behavior()` members of `os_collection`, `os_Collection`, and their subtypes. Indicates that the collection allows duplicate insertions of elements, and increments the count of each element by 1 with each insertion.

os_collection::allow_nulls

Possible element of the bit-wise disjunction return value of the `os_collection::get_behavior()` members of `os_collection`, `os_Collection`, and their subtypes. Indicates that the collection allows the insertion of null pointers.

os_collection::cardinality()

```
os_unsigned_int32 cardinality() const;
```

Returns the sum of the counts of each element of the specified collection.

os_collection::cardinality_estimate()

```
os_unsigned_int32 cardinality_estimate() const;
```

Returns an estimate of a collection's cardinality. This is an `O(1)` operation in the size of the collection. This function returns the cardinality as of the last call to `os_collection::update_cardinality()`; or, for collections that maintain cardinality, the actual cardinality is returned. Also, see `os_Dictionary::os_Dictionary()` on page 123 and `os_Dictionary::os_Dictionary()` on page 123.

os_collection::cardinality_is_maintained()

```
os_int32 cardinality_is_maintained() const;
```

Returns nonzero if the collection maintains cardinality; returns `0` otherwise. Also, see `os_Dictionary::os_Dictionary()` on page 123 and `os_Dictionary::os_Dictionary()` on page 123.

os_collection::clear()

```
void clear();
```

Removes all elements of the specified collection.

os_collection::contains()

```
os_int32 contains(const void*) const;
```

Returns a nonzero `os_int32` if the specified `void*` is an element of the specified collection, and `0` otherwise.

os_collection::count()

```
os_int32 count(const void*) const
```

Returns the number of occurrences (possibly 0) of the specified `void*` in the collection for which the function was called.

os_collection::empty()

```
os_int32 empty();
```

Returns a nonzero `os_int32` if the specified collection is empty, and `0` otherwise.

os_collection::EQ

Possible return value of the user-supplied rank functions and possible argument to `os_coll_range` constructors, signifying *equal*.

os_collection::GE

Possible argument to `os_coll_range` constructors, signifying *greater than or equal to*.

os_collection::GT

Possible return value of the user-supplied rank functions, and possible argument to `os_coll_range` constructors, signifying *greater than*.

os_collection::get_behavior()

```
os_unsigned_int32 get_behavior() const;
```

Returns a bit pattern indicating the specified collection's behavior. The return value is a bit-wise disjunction of enumerators indicating all the properties of the collection. For information about the enumerators, see

- `os_collection::allow_duplicates`
- `os_collection::allow_nulls`
- `os_collection::maintain_order`

os_collection::initialize()

```
static void initialize();
```

Must be called before using any ObjectStore collection or relationship functionality. Calling `initialize()` initializes the collections facility for the entire process. Calling `initialize()` more than once has no effect.

os_collection::insert()

```
void insert(const void*);
```

Adds the specified `void*` to the collection for which the function was called. The behavior of `insert()` depends on the characteristics of the collection you are using:

- If the collection is ordered, the element is inserted at the end of the collection.
- If the collection disallows duplicates, and the specified `void*` is already present in the collection, the insertion is silently ignored.
- If the collection disallows nulls, and the specified `void*` is `0`, `err_coll_nulls` is signaled.

os_collection::insert_after()

```
void insert_after(const void*, const os_cursor&);
```

Adds the specified `void*` to the collection for which the function was called. The new element is inserted immediately after the element at which the specified cursor is positioned. The index of all elements after the new element increases by 1. The cursor must be a default cursor (that is, one that results from a constructor call with only a single argument). If the cursor is null, `err_coll_null_cursor` is signaled. If the cursor is invalid, `err_coll_illegal_cursor` is signaled. If the collection maintained internally by the cursor is not the same as the collection maintained by the dictionary, the `err_coll_cursor_mismatch` exception is signaled.

```
void insert_after(const void*, os_unsigned_int32);
```

Adds the specified `void*` to the collection for which the function was called. The new element is inserted after the position indicated by the `os_unsigned_int32`. The index of all elements after the new element increases by 1. If the index is not less than the collection's cardinality, `err_coll_out_of_range` is signaled.

The behavior of `insert_after()` (both signatures) depends on the characteristics of the collection you are using:

- If the collection is not ordered, `err_coll_not_supported` is signaled.
- If the collection disallows duplicates, and the specified `void*` is already present in the collection, the insertion is silently ignored.
- If the collection disallows nulls, and the specified `void*` is `0`, `err_coll_nulls` is signaled.
- If the collection is an array, all elements after the inserted element are pushed down.

os_collection::insert_before()

```
void insert_before(const void*, const os_cursor&);
```

Adds the specified `void*` to the collection for which the function was called. The new element is inserted immediately before the element at which the specified cursor is positioned. The index of all elements after the new element increases by 1. The cursor must be a default cursor (that is, one that results from a constructor call with only a single argument). If the cursor is null, `err_coll_null_cursor` is signaled. If the cursor is invalid, `err_coll_illegal_cursor` is signaled. If the collection maintained internally by the cursor is not the same as the collection maintained by the dictionary, the `err_coll_cursor_mismatch` exception is signaled.

```
void insert_before(const void*, os_unsigned_int32);
```

Adds the specified instance of `void*` to the collection for which the function was called. The new element is inserted immediately before the position indicated by the `os_unsigned_int32`. The index of all elements after the new element increases by 1. If the index is not less than the collection's cardinality, `err_coll_out_of_range` is signaled.

The behavior of `insert_before()` (both signatures) depends on the characteristics of the collection you are using:

- If the collection is not ordered, `err_coll_not_supported` is signaled.
- If the collection disallows duplicates, and the specified `void*` is already present in the collection, the insertion is silently ignored.
- If the collection disallows nulls, and the specified `void*` is `0`, `err_coll_nulls` is signaled.
- If the collection is an array, all elements after this element are pushed down.

os_collection::insert_first()

```
void insert_first(const void*);
```

Adds the specified `void*` to the beginning of the collection for which the function was called. The index of all elements after the new element increases by 1.

The behavior of `insert_first()` depends on the characteristics of the collection you are using:

- If the collection is not ordered, `err_coll_not_supported` is signaled.
- If the collection disallows duplicates, and the specified `void*` is already present in the collection, `err_coll_duplicates` is signaled.
- If the collection disallows nulls, and the specified `void*` is `0`, the insertion is silently ignored.
- If the collection is an array, all elements after this element are pushed down.

os_collection::insert_last()

```
void insert_last(const void*);
```

Adds the specified `void*` to the end of the collection for which the function was called.

The behavior of `insert_last()` depends on the characteristics of the collection you are using:

- If the collection is not ordered, `err_coll_not_supported` is signaled.
- If the collection disallows duplicates, and the specified `void*` is already present in the collection, `err_coll_duplicates` is signaled.
- If the collection disallows nulls, and the specified `void*` is `0`, the insertion is silently ignored.

os_collection::LE

Possible argument to `os_coll_range` constructors, signifying *less than or equal to*.

os_collection::LT

Possible return value of the user-supplied `rank()` functions, and possible argument to `os_coll_range` constructors, signifying *less than*.

os_collection::maintain_order

Possible element of the bit-wise disjunction return value of the `os_collection::get_behavior()` members of `os_collection`, `os_Collection`, and their subtypes. Indicates that the collection maintains its elements in their order of insertion. This order is used as the default iteration order, as well as the relevant order for the members `insert_after()`, `insert_before()`, `insert_first()`, `insert_last()`, `remove_at()`, `remove_first()`, `remove_last()`, `retrieve_first()`, `retrieve_last()`, and `replace_at()`.

os_collection::NE

Possible argument to `os_coll_range` constructors, signifying *not equal to*.

os_collection::only()

```
void* only() const;
```

Returns the only element of the specified collection. If the collection has more than one element, `err_coll_not_singleton` is signaled. If the collection is empty, `0` is returned.

os_collection::operator os_int32()

```
operator os_int32() const;
```

Returns a nonzero `os_int32` if the specified collection is not empty, and `0` otherwise.

os_collection::operator os_array&()

```
operator os_array&();
```

Returns an array with the same elements and behavior as the specified collection. An exception is signaled if the collection's behavior is incompatible with the required behavior of arrays.

os_collection::operator const os_array&()

```
operator const os_array&() const;
```

Returns a `const` array with the same elements and behavior as the specified collection. An exception is signaled if the collection's behavior is incompatible with the required behavior of arrays.

os_collection::operator os_bag&()

```
operator os_bag&();
```

Returns a bag with the same elements and behavior as the specified collection. An exception is signaled if the collection's behavior is incompatible with the required behavior of bags.

os_collection::operator const os_bag&()

```
operator const os_bag&() const;
```

Returns a `const` bag with the same elements and behavior as the specified collection. An exception is signaled if the collection's behavior is incompatible with the required behavior of bags.

os_collection::operator os_list&()

```
operator os_list&();
```

Returns a list with the same elements and behavior as the specified collection. An exception is signaled if the collection's behavior is incompatible with the required behavior of lists.

os_collection::operator const os_list&()

```
operator const os_list&() const;
```

Returns a `const` list with the same elements and behavior as the specified collection. An exception is signaled if the collection's behavior is incompatible with the required behavior of lists.

os_collection::operator os_set&()

```
operator os_set&();
```

Returns a set with the same elements and behavior as the specified collection. An exception is signaled if the collection's behavior is incompatible with the required behavior of sets.

os_collection::operator const os_set&()

```
operator const os_set&() const;
```

Returns a `const` set with the same elements and behavior as the specified collection. An exception is signaled if the collection's behavior is incompatible with the required behavior of sets.

os_collection::operator ==()

```
os_int32 operator ==(const os_collection &s) const;
```

Returns a nonzero value if and only if for each element in the `this` collection `count(element) == s.count(element)`, and both collections have the same cardinality. Note that the comparison does not take order into account.

```
os_int32 operator ==(const void* s) const;
```

Returns a nonzero value if and only if the collection contains *s* and nothing else.

os_collection::operator !=()

```
os_int32 operator !=(const os_collection &s) const;
```

Returns a nonzero value if and only if it is not the case both that (1) for each element in the `this` collection `count(element) == s.count(element)`, and (2) both collections have the same cardinality. Note that the comparison does not take order into account.

```
os_int32 operator !=(const void* s) const;
```

Returns a nonzero value if and only if it is not the case that the collection contains *s* and nothing else.

os_collection::operator <()

`os_int32 operator <(const os_collection &s) const;`

Returns a nonzero value if and only if for each element in the `this` collection `count(element) <= s.count(element)` and `cardinality()` < `s.cardinality()`.

`os_int32 operator <(const void* s) const;`

Returns a nonzero value if and only if the specified collection is empty.

os_collection::operator <=()

`os_int32 operator <=(const os_collection &s) const;`

Returns a nonzero value if and only if for each element in the `this` collection `count(element) <= s.count(element)`.

`os_int32 operator <=(const void* s) const;`

Returns a nonzero value if and only if the specified collection is empty or *e* is the only element in the collection.

os_collection::operator >()

`os_int32 operator >(const os_collection &s) const;`

Returns a nonzero value if and only if for each element of *s*, `count(element) >=s.count(element)` and `cardinality() > s.cardinality()`.

`os_int32 operator >(const void* s) const;`

Returns a nonzero value if and only if `count(s) >= 1 and cardinality() > 1`.

os_collection::operator >=()

`os_int32 operator >=(const os_collection &s) const;`

Returns a nonzero value if and only if for each element of *s*, `count(element) >=s.count(element)`.

`os_int32 operator >=(const void* s) const;`

Returns a nonzero value if and only if `count(s) >= 1`.

Assignment Operator Semantics

Assignment operator semantics are described for the following functions in terms of insert operations into the target collection. The actual implementation of the assignment might be different, while still maintaining the associated semantics.

os_collection::operator =()

```
os_collection &operator =(const os_collection &s);
```

Copies the contents of the collection *s* into the target collection and returns the target collection. The copy is performed by effectively clearing the target, iterating over the source collection, and inserting each element into the target collection. The iteration is ordered if the source collection is ordered. The target collection semantics are enforced as usual during the insertion process.

```
os_collection &operator =(const void* e);
```

Clears the target collection, inserts the element *e* into the target collection, and returns the target collection.

os_collection::operator |=()

```
os_collection &operator |=(const os_collection &s);
```

Inserts the elements contained in *s* into the target collection and returns the target collection.

```
os_collection &operator |=(const void* e);
```

Inserts the element *e* into the target collection and returns the target collection.

os_collection::operator |()

```
os_collection &operator |(const os_collection &s) const;
```

Copies the contents of `this` into a new collection, `c`, and then performs `c |= s`. The new collection, `c`, is then returned. If either operand allows duplicates or nulls, the result does. If both operands maintain order, the result does.

```
os_collection &operator |(const void *e) const;
```

Copies the contents of `this` into a new collection, `c`, and then performs `c |= e`. The new collection, `c`, is then returned. If `this` allows duplicates, maintains order, or allows nulls, the result does.

os_collection::operator &=()

```
os_collection &operator &=(const os_collection &s);
```

For each element in the target collection, reduces the count of the element in the target to the minimum of the counts in the source and target collections. If the collection is ordered and contains duplicates, it does so by retaining the appropriate number of leading elements. It returns the target collection.

```
os_collection &operator &=(const void* e);
```

If *e* is present in the target, converts the target into a collection containing just the element *e*. Otherwise, it clears the target collection. It returns the target collection.

os_collection::operator &()

`os_collection &operator &(const os_collection &s) const;`

Copies the contents of `this` into a new collection, `c`, and then performs `c &= s`. The new collection, `c`, is then returned. If either operand allows duplicates or nulls, the result does. If both operands maintain order, the result does.

`os_collection &operator &(const void *e) const;`

Copies the contents of `this` into a new collection, `c`, and then performs `c &= e`. The new collection, `c`, is then returned. If `this` allows duplicates, maintains order, or allows nulls, the result does.

os_collection::operator –=()

`os_collection &operator –=(const os_collection &s);`

For each element in the collection *s*, removes `s.count(e)` occurrences of the element from the target collection. If the collection is ordered, it is the first `s.count(e)` elements that are removed. It returns the target collection.

`os_collection &operator –=(const void* e);`

Removes the element *e* from the target collection. If the collection is ordered, it is the first occurrence of the element that is removed from the target collection. It returns the target collection.

os_collection::operator -()

`os_collection &operator -(const os_collection &s) const;`

Copies the contents of `this` into a new collection, `c`, and then performs `c –= s`. The new collection, `c`, is then returned. If either operand allows duplicates or nulls, the result does. If both operands maintain order, the result does.

`os_collection &operator -(const void *e) const;`

Copies the contents of `this` into a new collection, `c`, and then performs `c –= e`. The new collection, `c`, is then returned. If `this` allows duplicates, maintains order, or allows nulls, the result does.

os_collection::remove_first()

`os_int32 remove_first(const void*&);`

Removes the first element from the specified collection, if the collection is not empty. Returns a nonzero `os_int32` if the collection was not empty, and `0` otherwise; and modifies its argument to refer to the removed element. If the specified collection is not ordered, `err_coll_not_supported` is signaled.

`void* remove_first();`

Removes the first element from the specified collection; returns the removed element, or `0` if the collection was empty. Note that for collections that allow null elements, the significance of the return value can be ambiguous. The alternative overloading of `remove_first()`, above, can be used to avoid the ambiguity. If the specified collection is not ordered, `err_coll_not_supported` is signaled. If the collection is an array, all elements after this element are pushed up.

os_collection::remove_last()

```
os_int32 remove_last(const void*&);
```

Removes the last element from the specified collection, if the collection is not empty; returns a nonzero `os_int32` if the collection was not empty, and modifies its argument to refer to the removed element. If the specified collection is not ordered, `err_coll_not_supported` is signaled.

```
void* remove_last();
```

Removes the last element from the specified collection; returns the removed element, or `0` if the collection was empty. Note that for collections that allow null elements, the significance of the return value can be ambiguous. The alternative overloading of `remove_last()`, above, can be used to avoid the ambiguity. If the specified collection is not ordered, `err_coll_not_supported` is signaled.

os_collection::replace_at()

```
void* replace_at(const void*, const os_cursor&);
```

Returns the element at which the specified cursor is positioned, and replaces it with the specified `void*`. The cursor must be a default cursor (that is, one that results from a constructor call with only a single argument). If the cursor is null, `err_coll_null_cursor` is signaled. If the cursor is nonnull but not positioned at an element, `err_coll_illegal_cursor` is signaled. If the collection maintained internally by the cursor is not the same as the collection maintained by the dictionary, the `err_coll_cursor_mismatch` exception is signaled.

- If the cursor is null, `err_coll_null_cursor` is signaled.
- If the cursor is invalid, `err_coll_illegal_cursor` is signaled.
- If the collection is not ordered, `err_coll_not_supported` is signaled.

```
void* replace_at(const void*, os_unsigned_int32 position);
```

Returns the element with the specified position, and replaces it with the specified `void*`. If the position is not less than the collection's cardinality, `err_coll_out_of_range` is signaled. If the collection is not ordered, `err_coll_not_supported` is signaled.

os_collection::retrieve()

`void* retrieve(const os_cursor&) const;`

Returns the element at which the specified cursor is positioned. The cursor must be a default cursor (that is, one that results from a constructor call with only a single argument). If the cursor is null, `err_coll_null_cursor` is signaled. If the cursor is nonnull but not positioned at an element, `err_coll_illegal_cursor` is signaled. If the collection maintained internally by the cursor is not the same as the collection maintained by the dictionary, the `err_coll_cursor_mismatch` exception is signaled.

- If the cursor is null, `err_coll_null_cursor` is signaled.
- If the cursor is invalid, `err_coll_illegal_cursor` is signaled.
- If the collection is not ordered, `err_coll_not_supported` is signaled.

`void* retrieve(os_unsigned_int32 `*`position`*`) const;`

Returns the element with the specified position. If the position is not less than the collection's cardinality, `err_coll_out_of_range` is signaled. If the collection is not ordered, `err_coll_not_supported` is signaled.

os_collection::retrieve_first()

`void* retrieve_first() const;`

Returns the specified collection's first element, or `0` if the collection is empty. For collections that contain zeros, see the following overloading of this function. If the collection is not ordered, `err_coll_not_supported` is signaled.

`os_int32 retrieve_first(const void*&) const;`

Returns `0` if the specified collection is empty; returns a nonzero `os_int32` otherwise. Modifies the argument to refer to the collection's first element. If the collection is not ordered, `err_coll_not_supported` is signaled.

os_collection::retrieve_last()

`void* retrieve_last() const;`

Returns the specified collection's last element, or `0` if the collection is empty. For collections that contain zeros, see the following overloading of this function. If the collection is not ordered, `err_coll_not_supported` is signaled.

`os_int32 retrieve_last(const void*&) const;`

Returns `0` if the specified collection is empty; returns a nonzero `os_int32` otherwise. Modifies the argument to refer to the collection's last element. If the collection is not ordered, `err_coll_not_supported` is signaled.

os_collection::update_cardinality()

`os_unsigned_int32 update_cardinality();`

Updates the value returned by `os_collection::cardinality_estimate()`, by scanning the collection and computing the actual cardinality.

os_Cursor

```
template <class E>
class os_Cursor : public os_cursor
```

An instance of this class serves to record the state of an iteration by pointing to the current element of an associated collection. A cursor's associated collection is specified when the cursor is created. The user can position the cursor in a relative fashion (using `next()` and `previous()`) or in absolute fashion (using `first()` and `last()`). The current element is retrieved using the positioning functions or `retrieve()`.

You can allocate a cursor in either transient or persistent memory.

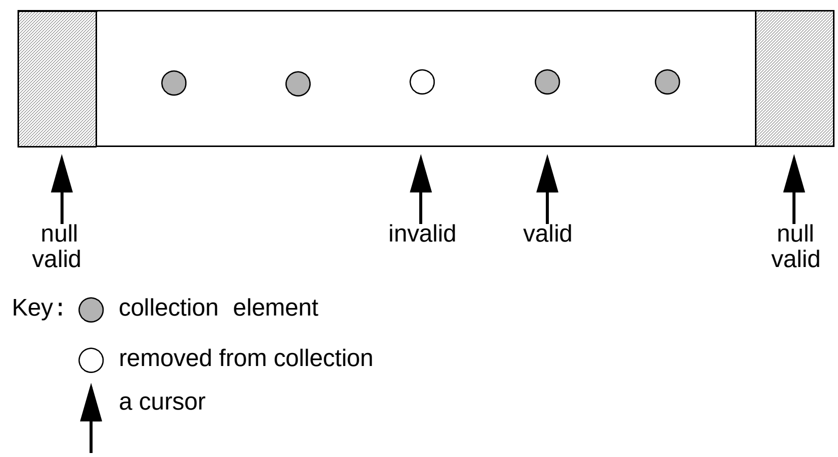
Every cursor has an associated ordering for the elements of its associated collection. This ordering can be the order in which elements appear in the collection (for ordered collections), an arbitrary order (for unordered collections), or the order in which elements appear in persistent memory (see `os_cursor::order_by_address` on page 116).

Upon creation of a persistent, ordered cursor, a write lock is acquired on segment 0 that effectively locks the entire database.

If a cursor is positioned at a collection's last element (in the cursor's associated ordering) and `next()` is performed on it, the cursor becomes *null*. Similarly, if a cursor is positioned at a collection's first element (in the cursor's associated ordering) and `previous()` is performed on it, the cursor becomes null. In other words, a cursor becomes null when it is either advanced past the last element or positioned before the first element. The function `os_Cursor::more()` returns a nonzero `os_int32` (true) if the specified cursor is not null, and returns `0` (false) if it is null.

If a cursor is positioned at an element of a collection, and then that element is removed from the collection, the cursor becomes *invalid*. Repositioning such a cursor has undefined results unless the flag `os_cursor::update_insensitive` was passed to the cursor constructor when the cursor was created. The function `os_Cursor::valid()` returns nonzero (true) if the specified cursor is valid, and returns `0` (false) if it is invalid.

The states *null* and *invalid* are mutually exclusive.



The class `os_Cursor` is *parameterized*, with a parameter indicating the element type of the associated collection — if an attempt is made to associate a cursor with a collection whose element type does not match the cursor's parameter, a compile-time error results. (For the nonparameterized version of this class, see `os_cursor` on page 114.) The parameter `E` occurs in the signatures of some of the functions described below. The parameter is used by the compiler to detect type errors.

os_Cursor::first()

```
E first();
```

Locates the specified cursor at the first element, in the cursor's associated ordering, of the cursor's associated collection. The first element is returned. If the collection is empty, the cursor is set to null and `0` is returned.

os_Cursor::insert_after()

```
void insert_after(const E p) const;
```

Inserts *p* into the cursor's associated collection immediately after the cursor's current location. If performed on a null cursor, `err_coll_null_cursor` is signaled. If the collection is an array, all elements after this one being inserted are pushed down.

os_Cursor::insert_before()

```
void insert_before(const E p) const;
```

Inserts *p* into the cursor's associated collection immediately before the cursor's current location. If performed on a null cursor, `err_coll_null_cursor` is signaled. If the collection is an array, all elements after this one being inserted are pushed down.

os_Cursor::last()

```
E last();
```

Locates the specified cursor at the last element, in the cursor's associated ordering, of the cursor's associated collection. The last element is returned. If the collection is empty, the cursor is set to null and `0` is returned.

os_Cursor::more()

```
os_int32 more();
```

Returns a nonzero `os_int32` (true) if the specified cursor is not null, that is, if the cursor is located at an element of the specified set or is invalid. The function returns `0` (false) otherwise.

os_Cursor::next()

```
E next();
```

Advances the specified cursor to the immediate next element of the cursor's associated collection, according to the cursor's associated ordering. The next element is returned. If there is no next element, or if the set is empty, the cursor is set to null and `0` is returned. If the cursor is null, a run-time error is signaled.

os_Cursor::null()

```
os_int32 null();
```

Returns a nonzero `os_int32` (true) if the specified cursor is null. The function returns `0` (false) if the cursor is located at an element of the specified set or is invalid. Inherited from `os_cursor`.

os_Cursor::os_Cursor()

```
os_Cursor<E> (
  const os_collection & coll,
  os_int32 os_cursor_enums = 0
);
```

Constructs a cursor associated with *`coll`*. If the collection is not ordered, the cursor's associated order is arbitrary, unless *`os_cursor_enums`* is `os_cursor::order_by_ address`, in which case the cursor's associated order is the order in which elements appear in persistent memory. If the collection is ordered and *`os_cursor_enums`* is `0`, the cursor's associated order is the order in which elements appear in the collection.

If you update a collection while traversing it without using an update-insensitive cursor, the results of the traversal are undefined.

If *`os_cursor_enums`* is `os_cursor::optimized`, it specifies that the cursor is transient. This option improves the performance of transient cursors that are used to iterate over persistent collections. This option is ignored if the cursor is persistent or restricted.

If *`os_cursor_enums`* is `os_cursor::order_by_address`, the cursor's associated order is the order in which elements appear in persistent memory. If you dereference each collection element as you retrieve it, and the objects pointed to by collection elements do not all fit in the client cache at once, this order can dramatically reduce paging overhead. An order-by-address cursor is update insensitive.

If *os_cursor_enums* is `os_cursor::update_insensitive`, the collection supports updates to it during traversal. The traversal visits exactly the elements of the collection at the time the cursor was bound. No insertions or removals performed during the traversal are reflected in the traversal.

If *os_cursor_enums* is `0`, the default, the cursor does not support updates to its associated collection during iteration.

```
os_Cursor<E>(
  const os_Collection<E> & coll,
  _Rank_fcn rfcn,
  os_int32 os_cursor_enums = 0
);
```

An *_Rank_fcn* is a rank function for the element type of *coll*. Iteration using that cursor follows the order determined by the specified rank function. The rank function must be registered with the `os_index_key` macro. Rank-function-based cursors are update insensitive.

```
os_Cursor<E> (
  const os_Collection<E> & coll,
  const char *typename,
  os_int32 os_cursor_enums = 0
);
```

typename is the name of the element type. Iteration using that cursor follows the order determined by the element type's rank function. The rank function must be registered with the `os_index_key` macro. Rank-function-based cursors are update insensitive.

os_Cursor::owner()

```
const os_collection *owner() const;
```

Returns a pointer to the specified cursor's associated collection. Inherited from `os_cursor`.

```
os_collection *owner();
```

Returns a pointer to the specified cursor's associated collection. Inherited from `os_cursor`.

os_Cursor::previous()

```
E previous();
```

Moves the specified cursor to the immediate previous element of the cursor's associated collection, according to the cursor's associated ordering. If there is no previous element, or if the collection is empty, the cursor is set to null and `0` is returned. If the cursor is null, a run-time error is signaled.

os_Cursor::rebind()

```
void rebind(const os_Collection<E>&);
```

Associates the specified cursor with the specified collection, positioning the cursor at the collection's first element. If the collection is empty, the cursor is not valid.

```
void rebind(const os_collection &, _Rank_Fcn);
```

Associates the specified cursor with the specified collection, positioning the cursor at the collection's first element according to the ordering provided by the rank function. If the collection is empty, the cursor is not valid.

os_Cursor::remove_at()

```
void remove_at() const;
```

Removes that element of the cursor's associated collection at which the specified cursor is currently located. If performed on a null or invalid cursor, `err_coll_null_cursor` is signaled. If the collection is an array, all elements after this one are pushed up.

os_Cursor::retrieve()

```
E retrieve();
```

Returns the element of the specified cursor's associated collection at which the specified cursor is currently located. A run-time error is signaled if the cursor is not located at an element of the set.

os_Cursor::valid()

```
os_int32 valid();
```

Returns a nonzero `os_int32` (true) if the specified cursor is null or is located at an element of the associated collection. The function returns `0` (false) if the cursor was located at an element that has been removed. Inherited from `os_cursor`.

os_Cursor::~os_Cursor()

```
void ~os_Cursor();
```

Breaks the association between the cursor and its associated collection.

os_cursor

An instance of this class serves to record the state of an iteration by pointing to the current element of an associated collection. A cursor's associated collection is specified when the cursor is created. The user can position the cursor in a relative fashion (using `next()` and `previous()`) or in absolute fashion (using `first()` and `last()`). The current element is retrieved using the positioning functions or `retrieve()`.

You can allocate a cursor in either transient or persistent memory.

Every cursor has an associated ordering for the elements of its associated collection. This ordering can be the order in which elements appear in the collection (for ordered collections), an arbitrary order (for unordered collections), or the order in which elements appear in persistent memory (see `os_cursor::order_by_address` on page 116).

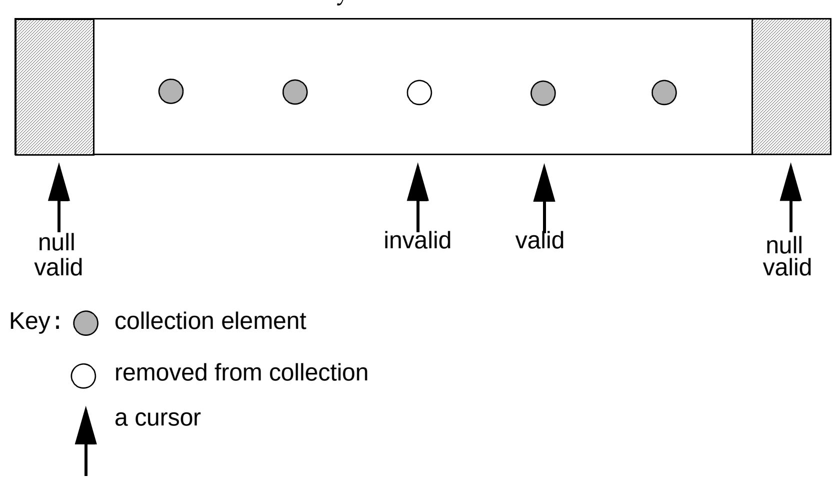
Upon creation of a persistent, ordered cursor, a write lock is acquired on segment 0 that effectively locks the entire database.

If a cursor is positioned at a collection's last element (in the cursor's associated ordering) and `next()` is performed on it, the cursor becomes *null*. Similarly, if a cursor is positioned at a collection's first element (in the cursor's associated ordering) and `previous()` is performed on it, the cursor becomes null. In other words, a cursor becomes null when it is either advanced past the last element or positioned before the first element. The function `os_cursor::more()` returns a nonzero `os_int32` (true) if the specified cursor is not null, and returns `0` (false) if it is null.

If a cursor is positioned at an element of a collection, and then that element is removed from the collection, the cursor becomes *invalid*. Repositioning such a cursor has undefined results. The function `os_cursor::valid()` returns nonzero (true) if the specified cursor is valid, and returns `0` (false) if it is invalid.

Valid and invalid cursors

The states *null* and *invalid* are mutually exclusive.



os_cursor::first()

```
void *first();
```

Locates the specified cursor at the first element of the cursor's associated collection, according to the cursor's associated ordering. The first element is returned. If the collection is empty, the cursor is set to null and `0` is returned.

os_cursor::insert_after()

```
void insert_after(const void *p) const;
```

Inserts *p* into the cursor's associated collection immediately after the cursor's current location. If performed on a null cursor, `err_coll_null_cursor` is signaled. If the collection is an array, all elements after this one being inserted are pushed down.

os_cursor::insert_before()

```
void insert_before(const void *p) const;
```

Inserts *p* into the cursor's associated collection immediately before the cursor's current location. If performed on a null cursor, `err_coll_null_cursor` is signaled. If the collection is an array, all elements after this element are pushed down.

os_cursor::last()

```
void *last();
```

Locates the specified cursor at the last element of the cursor's associated collection, according to the cursor's associated ordering. The last element is returned. If the collection is empty, the cursor is set to null and `0` is returned.

os_cursor::more()

```
os_int32 more();
```

Returns a nonzero `os_int32` (true) if the specified cursor is not null, that is, if the cursor is located at an element of the specified set or is invalid. The function returns `0` (false) otherwise.

os_cursor::next()

```
void *next();
```

Advances the specified cursor to the immediate next element of the cursor's associated collection, according to the cursor's associated ordering. The next element is returned. If there is no next element, or if the set is empty, the cursor is set to null and `0` is returned. If the cursor is null, a run-time error is signaled.

os_cursor::null()

```
os_int32 null();
```

Returns a nonzero `os_int32` (true) if the specified cursor is null. The function returns `0` (false) if the cursor is located at an element of the specified set or is invalid.

os_cursor::optimized

Possible argument to the `os_cursor` constructor, indicating that the cursor is transient. This option improves the performance of transient cursors that are used to iterate over persistent collections. This option is ignored if the cursor is persistent or restricted.

os_cursor::order_by_address

Possible argument to `os_cursor` or `os_Cursor` constructor, indicating that the cursor's associated ordering is the order in which elements appear in persistent memory.

If you dereference each collection element as you retrieve it, and the objects pointed to by collection elements do not all fit in the client cache at once, this order can dramatically reduce paging overhead. An order-by-address cursor is update insensitive.

os_cursor::os_cursor()

```
os_cursor(
  const os_collection & coll,
  os_int32 os_cursor_enums = 0
);
```

Constructs a cursor associated with *`coll`*. If the collection is not ordered, the cursor's associated order is arbitrary, unless *`os_cursor_enums`* is `os_cursor::order_by_address`, in which case the cursor's associated order is the order in which elements appear in persistent memory. If the collection is ordered and *`os_cursor_enums`* is `0`, the cursor's associated order is the order in which elements appear in the collection.

If you update a collection while traversing it without using an update-insensitive cursor, the results of the traversal are undefined.

If *`os_cursor_enums`* is `os_cursor::optimized`, it specifies that the cursor is transient. This option improves the performance of transient cursors that are used to iterate over persistent collections. This option is ignored if the cursor is persistent or restricted.

If *`os_cursor_enums`* is `os_cursor::order_by_address`, the cursor's associated order is the order in which elements appear in persistent memory. If you dereference each collection element as you retrieve it, and the objects pointed to by collection elements do not all fit in the client cache at once, this order can dramatically reduce paging overhead. An order-by-address cursor is update insensitive.

If *`os_cursor_enums`* is `os_cursor::update_insensitive`, the collection supports updates to it during traversal. The traversal visits exactly the elements of the collection at the time the cursor was bound. No insertions or removals performed during the traversal are reflected in the traversal.

If *`os_cursor_enums`* is `0`, the cursor does not support updates to its associated collection during iteration.

for this overloading of `os_cursor()`, if `os_cursor::update_insensitive` is specified for the *`os_cursor_enums`* argument, it is ignoredos_cursor(

```
  const os_dictionary & coll,
  const os_coll_range &range,
  os_int32 os_cursor_enums = 0
);
```

Constructs a cursor that can be used to traverse ordered dictionaries. A traversal with this cursor visits only those collection elements whose key satisfies `range`. The order of iteration is all the elements at a key value, followed by all the elements at the next key value, and so on. The order of the elements at each key value is arbitraryfor this overloading of `os_cursor()`, if `os_cursor::update_insensitive` is specified for the *`os_cursor_enums`* argument, it is ignored.

Copying a cursor

```
os_cursor (const os_cursor &c);
```

Constructs a new cursor by copying the contents of the cursor specified by *`c`*.

os_cursor::owner()

```
const os_collection *owner() const;
```

Returns a pointer to the specified cursor's associated collection.

```
os_collection *owner();
```

Returns a pointer to the specified cursor's associated collection.

os_cursor::previous()

```
void *previous();
```

Moves the specified cursor to the immediate previous element of the cursor's associated collection, according to the cursor's associated ordering. If there is no previous element, or if the collection is empty, the cursor is set to null and `0` is returned. If the cursor is null, a run-time error is signaled.

os_cursor::rebind()

```
void rebind(const os_collection&);
```

Associates the specified cursor with the specified collection, positioning the cursor at the collection's first element. If the collection is empty, the cursor is not valid.

```
void rebind(const os_collection&, _Rank_fcn);
```

Associates the specified cursor with the specified collection, positioning the cursor at the collection's first element according to the rank function. If the collection is empty, the cursor is not valid.

os_cursor::remove_at()

```
void remove_at() const;
```

Removes that element of the cursor's associated collection at which the specified cursor is currently located. If performed on a null or invalid cursor, `err_coll_null_cursor` is signaled. If the collection is an array, all elements after this element are pushed up.

os_cursor::retrieve()

```
void *retrieve();
```

Returns the element of the specified cursor's associated collection at which the specified cursor is currently located. A run-time error is signaled if the cursor is not located at an element of the collection.

os_cursor::update_insensitive

Possible argument to `os_cursor` or `os_Cursor` constructor, indicating that the collection supports updates during traversal of the cursor. The traversal visits exactly the elements of the collection at the time the cursor was bound.

os_cursor::valid()

```
os_int32 valid();
```

Returns a nonzero `os_int32` (true) if the specified cursor is null or is located at an element of the associated collection. The function returns `0` (false) if the cursor was located at an element that has been removed.

os_cursor::~os_cursor()

```
void ~os_cursor();
```

Breaks the association between the cursor and its associated collection.

os_Dictionary

```
template <class K, class E>
class os_Dictionary<K, E> : public os_Collection<E>
```

A dictionary is a collection that can be either ordered or unordered, allows duplicate elements, and associates a key with each element. The key can be a value of any C++ fundamental type or user-defined class. If the key is a pointer, it must be a `void*`. When you insert an element into a dictionary, you specify the key along with the element. You can retrieve an element with a given key or retrieve those elements whose keys fall within a given range. `os_Dictionary` inherits from `os_collection`.

Dictionaries are always implemented as B-trees or hash tables, so look-up of elements based on their keys is efficient.

If you use persistent dictionaries, you must call the macro `OS_MARK_DICTIONARY()` in your source file for each key-type/element-type pair that you use. If you are using only transient dictionaries, call the macro `OS_TRANSIENT_DICTIONARY()` in your source file.

The element type of any instance of `os_Dictionary` must be a pointer type.

Required attributes

Requirements for classes used as keys are listed below.

- The class must have a constructor that takes no arguments, and the destructor must be able to handle a class that is constructed with the no-argument constructor.
- The class must have an `operator=()` defined.
- The class must have a destructor, and the destructor must delete any storage allocated either by the no-argument constructor or by `operator=.` The destructor must also set any pointers in the object to `0`. The destructor should also be prepared to handle the deletion of the object where all the data is zeroed out.
- You must define and register (using `os_index_key`) rank/hash functions for the class type.

These requirements apply only to default unordered dictionaries. For dictionaries that are ordered, the key is initialized using `memcpy()`.

For integer keys, specify one of the following as the key type:

- `os_int32` (a signed 32-bit integer)
- `os_unsigned_int32` (an unsigned 32-bit integer)
- `os_int16` (a signed 16-bit integer)
- `os_unsigned_int16` (an unsigned 16-bit integer)

Use the type `void*` for pointer keys other than `char*` keys.

For `char[]` keys, use the parameterized type `os_char_array<S>`, where the actual parameter is an integer literal indicating the size of the array in bytes.

The key type `char*` is treated as a class whose rank and hash functions are defined in terms of `strcmp( )` or `strcoll( )`. For example:

```
a_dictionary.pick("Smith")
```

returns an element of `a_dictionary` whose key is the string “Smith” (that is, whose key, `k`, is such that `strcmp(k, "Smith")` is `0`).

If a dictionary’s key type is `char*` and it is unordered, the dictionary makes its own copies of the character array upon insertion. If the key type is `char*` and the dictionary has the behavior `maintain_key_order`, it points to the string rather than makes a copy of it. If the dictionary does not allow duplicate keys, you can significantly improve performance by using the type `os_char_star_nocopy` as the key type. With this key type, the dictionary copies the pointer to the array and not the array itself. You can freely pass `char*`s to this type.

Note that you cannot use `os_char_star_nocopy` with dictionaries that allow duplicate keys.

Required header files

Any program using dictionaries must include the header files <os_pse/ostore.hh> followed by <os_pse/coll.hh>. In addition, your program will require the inclusion of <os_pse/coll/dict_pt.hh> or <os_pse/coll/dict_pt.cc>.

If your program instantiates a template, include `dict_pt.cc` at the point where you instantiate the template. If you are using the template, but not instantiating it, include `dict_pt.hh`. Since `dict_pt.cc` includes `dict_pt.hh`, you do not need both. You have to include `dict_pt.cc` because it contains the bodies of the functions declared in `dict_pt.hh`.

Tables of member functions and enumerators

The first of the following tables lists the member functions that can be performed on instances of `os_Dictionary`. The second table lists the enumerators inherited by `os_Dictionary` from `os_collection`. Many functions are also inherited by `os_Dictionary` from `os_Collection` or `os_collection`. The full explanation of each inherited function or enumerator appears in the entry for the class from which it is inherited. The full explanation of each function defined by `os_Dictionary` appears in this entry, after the tables. In each case, the *Defined By* column gives the class whose entry contains the full explanation.

Name	*Arguments*	*Returns*	*Defined By*
`cardinality`	`( ) const`	`os_unsigned_int32`	`os_collection`
`clear`	`( )`	`void`	`os_collection`

`contains`	`( const K &key_ref,`
	`const E element ) const`
	`( const K *key_ptr,`
	`const E element ) const`
		`os_int32`
		
		`os_int32`
			`os_Dictionary`

`count`	`( const E ) const`	`os_int32`	`os_Collection`
`count_values`	`( const K &key_ref ) const` `( const K * key_ptr ) const`	`os_unsigned_int32` `os_unsigned_int32`	`os_Dictionary`
`default_behavior` `(static)`	`( )`	`os_unsigned_int32`	`os_Dictionary`

Name	*Arguments*	*Returns*	*Defined By*
empty	( )	os_int32	os_collection
get_behavior	( ) const	os_unsigned_int32	os_collection
insert	(const K &key_ref, const E element) (const K *key_ptr, const E element)	void void	os_Dictionary
only	( ) const	E	os_Collection
os_Dictionary	(os_unsigned_int32 expected_card = 10, os_unsigned_int32 behavior_ enums = 0)		os_Dictionary
pick	(const K &key_ref) const (const K *key_ptr) const ( ) const	E E E	os_Dictionary
remove	(const K &key_ref, const E element) (const K *key_ptr, const E element) (const E element)	void void os_int32	os_Dictionary
remove_value	(const K &key_ref, const E os_unsigned_int32 n = 1) (const K *key_ptr, os_unsigned_int32 n = 1)	E E	os_Dictionary
retrieve	(const os_cursor&) const	E	os_Dictionary
retrieve_key	(const os_cursor&)	K* or K	os_Dictionary

os_Dictionary enumerators

The following table lists enumerators for the os_Dictionary class.

Name	*Inherited From*
allow_duplicates	os_collection
allow_nulls	os_collection
EQ	os_collection
GT	os_collection
LT	os_collection
maintain_cardinality	os_dictionary
maintain_key_order	os_dictionary
maintain_order	os_collection
no_dup_keys	os_dictionary
signal_dup_keys	os_dictionary

os_Dictionary::contains()

`os_boolean contains(const K &key_ref, const E element) const;`

Returns nonzero (true) if `this` contains an entry with the specified element and the key referred to by *`key_ref`*. If there is no such entry, `0` (false) is returned. This overloading of `contains()` differs from the next overloading only in that the key is specified with a reference instead of a pointer.

`os_boolean contains(const K *key_ptr, const E element) const;`

Returns nonzero (true) if `this` contains an entry with the specified element and the key pointed to by *`key_ptr`*. If there is no such entry, `0` (false) is returned. This overloading of `contains()` differs from the previous overloading only in that the key is specified with a pointer instead of a reference.

os_Dictionary::count_values()

`os_unsigned_int32 count_values(const K &key_ref) const;`

Returns the number of entries in `this` with the key referred to by *`key_ref`*. This overloading of `count_values()` differs from the next overloading only in that the key is specified with a reference instead of a pointer.

`os_unsigned_int32 count_values(const K *key_ptr) const;`

Returns the number of entries in `this` with the key pointed to by *`key_ptr`*. This overloading of `count_values()` differs from the previous overloading only in that the key is specified with a pointer instead of a reference.

os_Dictionary::insert()

`void insert(const K &key_ref, const E element);`

Inserts the specified element with the key referred to by *`key_ref`*. This overloading of `insert()` differs from the next overloading only in that the key is specified with a reference instead of a pointer.

Each insertion increases the collection’s cardinality by 1 and increases by 1 the count (or number of occurrences) of the inserted element in the collection, unless the dictionary already contains an entry that matches both the key and the element (in which case the insertion is silently ignored).

If you insert a null pointer (`0`), the exception `err_coll_nulls` is signaled.

For dictionaries with `signal_dup_keys` behavior, if an attempt is made to insert something with a key that already exists, `err_am_dup_key` is signaled.

`void insert(const K *key_ptr, const E element);`

Inserts the specified element with the key pointed to by *`key_ptr`*. This overloading of `insert()` differs from the previous overloading only in that the key is specified with a pointer instead of a reference.

os_Dictionary::os_Dictionary()

```
os_Dictionary(
  os_unsigned_int32 expected_cardinality = 10,
  os_unsigned_int32 behavior_enums = 0
);
```

Creates a new dictionary. The following paragraphs describe the arguments.

Presizing cardinality

By default, dictionaries are presized with an internal structure suitable for cardinality `10`. If you want a new dictionary presized for a different cardinality, specify the *`expected_cardinality`* argument. Specifying this argument allows the user to incur the cost of growing the internal structure at creation time rather than during the life of the dictionary.

Dictionary properties

Every dictionary has the following properties:

- Duplicate elements are allowed.
- Null pointers cannot be inserted.
- There is no guarantee that an element inserted or removed during a traversal will be visited later in the same traversal.
- Performing `pick()` on an empty result of querying the dictionary returns `0`.

By default, a new dictionary also has the following properties:

- Its elements have no intrinsic order.
- Duplicate keys are allowed; that is, two or more elements can have the same key.
- Range look-ups are not supported; that is, key order is not maintained.

You can enable or disable these last three properties in new dictionaries. To do so, specify the *`behavior_enums`* argument with a bit pattern that indicates the collection's properties. The bit pattern is obtained by forming the bit-wise disjunction (using the bit-wise OR operator) of the following enumerators:

- `os_Dictionary::signal_dup_keys`: Duplicate keys are not allowed; `err_coll_duplicate_key` is signaled if an attempt is made to establish two or more elements with the same key.
- `os_Dictionary::maintain_key_order`: Range look-ups are supported using `pick()` or restricted cursors. By default this implies `os_Dictionary::dont_maintain_cardinality`. You can mantain cardinality, but it is discouraged for dictionaries with large cardinality because performance is compromised.
- `os_Dictionary::maintain_cardinality`: For dictionaries that maintain key order, the `insert()` and `remove()` functions will update cardinality information. This enumerator will improve performance if the `cardinality()` function is used (which otherwise is an `O(n)` operation when cardinality is not maintained. However, maintaining cardinality can also cause contention in the dictionary header that can have an impact on large databases. For dictionaries that do not maintain key order, cardinality is always maintained but is done without causing contention in the dictionary header; in this case setting the `maintain_cardinality` enumerator is ignored.

These enumerators are instances of an enumeration defined in the scope of `os_Dictionary`. Each enumerator is associated with a different bit; including an enumerator in the disjunction sets its associated bit.

For large dictionaries that maintain key order, the enumerator `os_Dictionary::dont_maintain_cardinality` can also be used to reduce contention. When this enumerator is specified in the disjunction, `behavior_enums`, `insert()`, and `remove()` do not update cardinality information, avoiding contention in the collection header. This behavior can significantly improve performance for large dictionaries subject to contention. The disadvantage of this behavior is that `cardinality()` is an `O(n)` operation, requiring a scan of the whole dictionary.

For more information, see

- `os_collection::cardinality_estimate()` on page 98
- `os_collection::cardinality_is_maintained()` on page 98
- `os_collection::update_cardinality()` on page 108

```
os_Dictionary<K, E>::os_Dictionary<K, E>(
  os_unsigned_int32 expected_cardinality = 10,
  os_unsigned_int32 behavior_enums = 0
);
```

Creates a new dictionary. The key type is specified by the parameter `K` and the value type by the parameter `E`. If the key type is a pointer type, the parameter `K` should be `void*`. The arguments have the same meaning as for the nonparameterized constructor.

os_Dictionary::pick()

```
E pick(const K &key_ref) const;
```

Returns an element of `this` that has the value of the key referred to by the value of *`key_ref`*. If there is more than one such element, an arbitrary one is picked and returned. If there is no such element, `0` is returned. If the dictionary is empty, returns `0`.

```
E pick(const K *key_ptr) const;
```

Returns an element of `this` that has the value of the key pointed to by *`key_ptr`*. If there is more than one such element, an arbitrary one is picked and returned. If there is no such element, `0` is returned. If the dictionary is empty, returns `0`.

```
E pick() const;
```

Picks an arbitrary element of `this` and returns it. If the dictionary is empty, `0` is returned.

os_Dictionary::remove()

```
void remove(const K &key_ref, const E element);
```

Removes the dictionary entry with the element `element` at the value of the key referred to by `key_ref`. This overloading of `remove()` differs from the next overloading only in that the key is specified with a reference instead of a pointer. If removing this element leaves no other elements at this key value, the key is removed and deleted.

If there is no such entry, the dictionary remains unchanged. If there is such an entry, the collection’s cardinality decreases by 1 and the count (or number of occurrences) of the removed element in the collection decreases by 1.

```
void remove(const K *key_ptr, const E element);
```

Removes the dictionary entry with the element `element` and the key pointed to by `key_ptr`. This overloading of `remove()` differs from the previous overloading only in that the key is specified with a pointer instead of a reference. If removing this element leaves no other elements at this key value, the key is removed and deleted.

```
os_int32 remove(const E element)
```

Removes one dictionary entry specified by `element`. The element removed may have any key value. If removing this element leaves no other elements at its corresponding key value, the corresponding key value is removed and deleted.

If there is no such entry, the dictionary remains unchanged. If there is such an entry, the collection's cardinality decreases by 1 and the count (or number of occurrences) of the removed element in the collection decreases by 1.

Returns a nonzero `os_int32` if an element was removed, and 0 otherwise.

os_Dictionary::remove_value()

```
E remove_value(const K &key_ref, os_unsigned_int32 n = 1);
```

Removes *n* dictionary entries with the value of the key referred to by `key_ref`. If there are fewer than *n*, all entries in the dictionary with that key are removed. If there is no such entry, the dictionary remains unchanged.

This overloading of `remove_value()` differs from the next overloading only in that the key is specified with a reference instead of a pointer.

For each entry removed, the collection’s cardinality decreases by `1` and the count (or number of occurrences) of the removed element in the collection decreases by 1. If removing this element leaves no other elements at this key value, the key is removed and deleted.

```
void remove_value(const K *key_ptr, os_unsigned_int32 n = 1);
```

Removes *n* dictionary entries with the value of the key pointed to by `key_ptr`. This overloading of `remove_value()` differs from the previous overloading only in that the key is specified with a pointer instead of a reference. If removing this element leaves no other elements at this key value, the key is removed and deleted.

os_Dictionary::retrieve()

```
E retrieve(const os_cursor&) const;
```

Returns the element of `this` at which the specified cursor is located. If the cursor is null, `err_coll_null_cursor` is signaled. If the cursor is invalid, `err_coll_illegal_cursor` is signaled. If the collection maintained internally by the cursor is not the same as the collection maintained by the dictionary, the `err_coll_cursor_mismatch` exception is signaled.

os_Dictionary::retrieve_key()

```
const K *retrieve_key(const os_cursor&) const;
```

```
const K retrieve_key(const os_cursor&) const;
```

Returns a pointer to a dictionary key unless the key is a `char*` or `void*` in which case `retrieve_key()` returns the key itself. You must not modify this key. Like all collection functions that take cursor arguments, this function works only with vanilla cursors — that is, cursors that were not created with a cursor option or rank function. If the collection maintained internally by the cursor is not the same as the collection maintained by the dictionary, the `err_coll_cursor_mismatch` exception is signaled.

os_List

```
template <class E>
class os_List : public os_Collection<E>
```

A list is an ordered collection. As with other ordered collections, list elements can be inserted, removed, replaced, or retrieved based on a specified numerical index or based on the position of a specified cursor.

Lists allow duplicates and disallow null elements.

If an element is inserted or removed from an `os_List`, all other elements are either pushed up or down with respect to their ordinal index in the list.

The class `os_List` is *parameterized*, with a parameter for constraining the type of values allowable as elements (for the nonparameterized version of this class, see `os_list` on page 132). The element type parameter, `E`, occurs in the signatures of some of the functions described below. The parameter is used by the compiler to detect type errors.

It is possible to optimize `os_List` for maximum performance, using `os_nList`; see `os_nList and os_nlist` on page 137.

The element type of any instance of `os_List` must be a pointer type.

You must mark parameterized collections types in the schema source file.

Tables of member functions and enumerators

The first of the following tables lists the member functions that can be performed on instances of `os_List`. The second table lists the enumerators inherited by `os_List` from `os_collection`. Many functions are also inherited by `os_List` from `os_Collection` or `os_collection`. The full explanation of each inherited function or enumerator appears in the entry for the class from which it is inherited. The full explanation of each function defined by `os_List` appears in this entry, after the tables. In each case, the *Defined By* column gives the class whose entry contains the full explanation.

Name	*Arguments*	*Returns*	*Defined By*
`cardinality`	`( ) const`	`os_int32`	`os_collection`
`clear`	`( )`	`void`	`os_collection`
`contains`	`( const E ) const`	`os_int32`	`os_Collection`
`count`	`( const E ) const`	`os_int32`	`os_Collection`
`empty`	`( )`	`os_int32`	`os_collection`
`get_behavior`	`( ) const`	`os_unsigned_int32`	`os_collection`
`insert`	`( const E )`	`void`	`os_Collection`
`insert_after`	`( const E, const os_Cursor<E>& )`	`void`	`os_Collection`
	`( const E, os_unsigned_int32 )`	`void`	`os_Collection`

Name	*Arguments*	*Returns*	*Defined By*

insert_before	(const E,
	  const os_Cursor<E>&)
	(const E,
	  os_unsigned_int32)
		void
		void
			os_Collection

insert_first	(const E)	void	os_Collection
insert_last	(const E)	void	os_Collection
only	( ) const	E	os_Collection
operator os_Array<E>&	( )		os_Collection
operator const os_Array<E>&	( ) const		os_Collection
operator os_array&	( )		os_collection
operator const os_array&	( ) const		os_collection
operator os_Bag<E>&	( )		os_Collection
operator const os_Bag<E>&	( ) const		os_Collection
operator os_bag&	( )		os_collection
operator const os_bag&	( ) const		os_collection
operator os_list&	( )		os_collection
operator const os_list&	( ) const		os_collection
operator os_Set<E>&	( )		os_Collection
operator const os_Set<E>&	( ) const		os_Collection
operator os_set&	( )		os_collection
operator const os_set&	( ) const		os_collection
operator ==	(const os_Collection<E>&) const (const E) const	os_int32 os_int32	os_Collection
operator !=	(const os_Collection<E>&) const (const E) const	os_int32 os_int32	os_Collection
operator <	(const os_Collection<E>&) const (const E) const	os_int32 os_int32	os_Collection
operator <=	(const os_Collection<E>&) const (const E) const	os_int32 os_int32	os_Collection

<table>
<tr><th>Name</th><th>Arguments</th><th>Returns</th><th>Defined By</th></tr>
<tr><td>operator ></td><td>(const os_Collection<E>&) const
(const E) const</td><td>os_int32
os_int32</td><td>os_Collection</td></tr>
<tr><td>operator >=</td><td>(const os_Collection<E>&) const
(const E) const</td><td>os_int32
os_int32</td><td>os_Collection</td></tr>
<tr><td>operator =</td><td>(const os_List<E>&) const
(const os_Collection<E>&) const
(const E) const</td><td>os_List<E>&
os_List<E>&
os_List<E>&</td><td>os_List</td></tr>
<tr><td>operator |=</td><td>(const os_Collection<E>&) const
(const E) const</td><td>os_List<E>&
os_List<E>&</td><td>os_List</td></tr>
<tr><td>operator |</td><td>(const os_Collection<E>&) const
(const E) const</td><td>os_Collection<E>&
os_Collection<E>&</td><td>os_Collection</td></tr>
<tr><td>operator &=</td><td>(const os_Collection<E>&) const
(const E) const</td><td>os_List<E>&
os_List<E>&</td><td>os_List</td></tr>
<tr><td>operator &</td><td>(const os_Collection<E>&) const
(const E) const</td><td>os_Collection<E>&
os_Collection<E>&</td><td>os_Collection</td></tr>
<tr><td>operator -=</td><td>(const os_Collection<E>&) const
(const E) const</td><td>os_List<E>&
os_List<E>&</td><td>os_List</td></tr>
<tr><td>operator -</td><td>(const os_Collection<E>&) const
(const E) const</td><td>os_Collection<E>&
os_Collection<E>&</td><td>os_Collection</td></tr>
<tr><td>os_List</td><td>( )
(os_int32 expected_size)
(const os_List<E>&)
(const os_Collection<E>&)</td><td></td><td>os_List</td></tr>
<tr><td>remove</td><td>(const E)</td><td>os_int32</td><td>os_Collection</td></tr>
<tr><td>remove_at</td><td>(const os_Cursor<E>&)
(os_unsigned_int32)</td><td>void
void</td><td>os_Collection</td></tr>
<tr><td>remove_first</td><td>(const E&)
( )</td><td>os_int32
E</td><td>os_Collection</td></tr>
<tr><td>remove_last</td><td>(const E&)
( )</td><td>os_int32
E</td><td>os_Collection</td></tr>
<tr><td>replace_at</td><td>(const E, const os_Cursor<E>&)
(const E, os_unsigned_int32)</td><td>E
E</td><td>os_Collection</td></tr>
<tr><td>retrieve</td><td>(os_unsigned_int32) const
(const os_Cursor<E>&) const</td><td>E
E</td><td>os_Collection</td></tr>
</table>

Name	*Arguments*	*Returns*	*Defined By*
retrieve_first	( ) const (const E&) const	E os_int32	os_Collection
retrieve_last	( ) const (const E&) const	E os_int32	os_Collection

os_List enumerators

The following table lists the enumerators inherited by `os_List` from `os_collection`.

Name	*Inherited From*
allow_duplicates	os_collection
allow_nulls	os_collection
EQ	os_collection
GT	os_collection
LT	os_collection
maintain_order	os_collection

Assignment Operator Semantics

Assignment operator semantics are described for the following functions in terms of insert operations into the target collection. The actual implementation of the assignment might be different, while still maintaining the associated semantics.

os_List::operator =()

```
os_List<E> &operator =(const os_List<E> &s);
```

Copies the contents of the collection *s* into the target collection and returns the target collection. The copy is performed by effectively clearing the target, iterating over the source collection (in order), and inserting each element into the target collection. The target collection semantics are enforced as usual during the insertion process.

```
os_List<E> &operator =(const os_Collection<E> &s);
```

Copies the contents of the collection *s* into the target collection and returns the target collection. The copy is performed by effectively clearing the target, iterating over the source collection (in order), and inserting each element into the target collection. The target collection semantics are enforced as usual during the insertion process.

```
os_List<E> &operator =( const E e);
```

Clears the target collection, inserts the element *e* into the target collection, and returns the target collection.

os_List::operator |=()

```
os_List<E> &operator |=(const os_Collection<E> &s);
```

Inserts the elements contained in *s* into the target collection and returns the target collection.

```
os_List<E> &operator |=(const E e);
```

Inserts the element *e* into the target collection and returns the target collection.

os_List::operator &=()

```
os_List<E> &operator &=(const os_Collection<E> &s);
```

For each element in the target collection, reduces the count of the element in the target to the minimum of the counts in the source and target collections. It does so by retaining the appropriate number of leading elements. It returns the target collection.

```
os_List<E> &operator &=(const E e);
```

If *e* is present in the target, converts the target into a collection containing just the element *e*. Otherwise, it clears the target collection. It returns the target collection.

os_List::operator –=()

```
os_List<E> &operator –=(const os_Collection<E> &s);
```

For each element in the collection *s*, removes `s.count(e)` occurrences of the element from the target collection. The first `s.count(e)` elements are removed. It returns the target collection.

```
os_List<E> &operator –=(const E e);
```

Removes the element *e* from the target collection. The first occurrence of the element is removed from the target collection. It returns the target collection.

os_List::os_List()

```
os_List();
```

Returns an empty list.

```
os_List(os_int32 expected_size, os_int32 behavior = 0);
```

Returns an empty list whose initial implementation is based on the expectation that the *expected_size* argument approximates the usual cardinality of the list, once the list has been loaded with elements.

```
os_List(const os_List<E> &coll);
```

Returns a list that results from assigning the specified list to an empty list.

```
os_List(const os_Collection<E> &coll);
```

Returns a list that results from assigning the specified collection to an empty list.

os_list

`class os_list : public os_collection`

A list is an ordered collection. As with other ordered collections, list elements can be inserted, removed, replaced, or retrieved based on a specified numerical index or based on the position of a specified cursor.

The class `os_list` is nonparameterized. For the parameterized version of this class, see `os_List` on page 127.

Lists allow duplicates and disallow null elements.

If an element is inserted or removed from an `os_list`, all other elements are either pushed up or down with respect to their ordinal index in the list.

It is possible to optimize `os_list` for maximum performance, using `os_nlist`; see `os_nList and os_nlist` on page 137.

The element type of any instance of `os_list` must be a pointer type.

Tables of member functions and enumerators

The first of the following tables lists the member functions that can be performed on instances of `os_list`. The second table lists the enumerators inherited by `os_list` from `os_collection`. The full explanation of each inherited function or enumerator appears in the entry for the class from which it is inherited. The full explanation of each function defined by `os_list` appears in this entry, after the tables. In each case, the *Defined By* column gives the class whose entry contains the full explanation.

Name	*Arguments*	*Returns*	*Defined By*
cardinality	( ) const	os_int32	os_collection
clear	( )	void	os_collection
contains	(const void*) const		os_collection
count	(const void*) const	os_int32	
empty	( )	os_int32	os_collection
get_behavior	( ) const	os_unsigned_int32	os_collection
insert	(const void*)	void	os_collection
insert_after	(const void*, const os_cursor&) (const void*, os_unsigned_int32)	void void	os_collection
insert_before	(const void*, const os_cursor&) (const void*, os_unsigned_int32)	void void	os_collection
insert_first	(const void*)	void	os_collection
insert_last	(const void*)	void	os_collection
only	( ) const	void*	os_collection

<table>
<tr><th>Name</th><th>Arguments</th><th>Returns</th><th>Defined By</th></tr>
<tr><td>operator os_bag&</td><td>( )</td><td></td><td>os_collection</td></tr>
<tr><td>operator const os_bag&</td><td>( ) const</td><td></td><td>os_collection</td></tr>
<tr><td>operator os_set&</td><td>( )</td><td></td><td>os_collection</td></tr>
<tr><td>operator const os_set&</td><td>( ) const</td><td></td><td>os_collection</td></tr>
<tr><td>operator ==</td><td>(const os_collection&) const
(const void*) const</td><td>os_int32
os_int32</td><td>os_collection</td></tr>
<tr><td>operator !=</td><td>(const os_collection&) const
(const void*) const</td><td>os_int32
os_int32</td><td>os_collection</td></tr>
<tr><td>operator <</td><td>(const os_collection&) const
(const void*) const</td><td>os_int32
os_int32</td><td>os_collection</td></tr>
<tr><td>operator <=</td><td>(const os_collection&) const
(const void*) const</td><td>os_int32
os_int32</td><td>os_collection</td></tr>
<tr><td>operator ></td><td>(const os_collection&) const
(const void*) const</td><td>os_int32
os_int32</td><td>os_collection</td></tr>
<tr><td>operator >=</td><td>(const os_collection&) const
(const void*) const</td><td>os_int32
os_int32</td><td>os_collection</td></tr>
<tr><td>operator =</td><td>(const os_list&) const
(const os_collection&) const
(const void*) const</td><td>os_list&
os_list&
os_list&</td><td>os_list</td></tr>
<tr><td>operator |=</td><td>(const os_collection&) const
(const void*) const</td><td>os_list&
os_list&</td><td>os_list</td></tr>
<tr><td>operator |</td><td>(const os_collection&) const
(const void*) const</td><td>os_collection
os_collection</td><td>os_collection</td></tr>
<tr><td>operator &=</td><td>(const os_collection&) const
(const void*) const</td><td>os_list&
os_list&</td><td>os_list</td></tr>
<tr><td>operator &</td><td>(const os_collection&) const
(const void*) const</td><td>os_collection
os_collection</td><td>os_collection</td></tr>
<tr><td>operator –=</td><td>(const os_collection&) const
(const void*) const</td><td>os_list&
os_list&</td><td>os_list</td></tr>
<tr><td>operator -</td><td>(const os_collection&) const
(const void*) const</td><td>os_collection
os_collection</td><td>os_collection</td></tr>
</table>

Name	*Arguments*	*Returns*	*Defined By*

`os_list`	`( )`
	`( os_int32 expected_size )`
	`( const os_list& )`
	`( const os_collection& )`
			`os_list`

`remove`	`( const void* )`	`os_int32`	`os_collection`
`remove_at`	`( const os_cursor& )` `( os_unsigned_int32 )`	`void` `void`	`os_collection`
`remove_first`	`( const void*& )` `( )`	`os_int32` `void*`	`os_collection`
`remove_last`	`( const void*& )` `( )`	`os_int32` `void*`	`os_collection`
`replace_at`	`( const void*, const os_cursor& )` `( const void*, os_unsigned_int32 )`	`void*` `void*`	`os_collection`
`retrieve`	`( os_unsigned_int32 ) const` `( const os_cursor& ) const`	`void*` `void*`	`os_collection`
`retrieve_first`	`( ) const` `( const void*& ) const`	`void*` `os_int32`	`os_collection`
`retrieve_last`	`( ) const` `( const void*& ) const`	`void*` `os_int32`	`os_collection`

os_list enumerators

The following table lists the enumerators inherited by `os_list` from `os_collection`.

Name	*Inherited From*
`allow_duplicates`	`os_collection`
`allow_nulls`	`os_collection`
`EQ`	`os_collection`
`GT`	`os_collection`
`LT`	`os_collection`
`maintain_order`	`os_collection`

Assignment Operator Semantics

Assignment operator semantics are described for the following functions in terms of insert operations into the target collection. The actual implementation of the assignment might be different, while still maintaining the associated semantics.

os_list::operator =()

`os_list &operator =(const os_collection &s);`

Copies the contents of the collection *s* into the target collection and returns the target collection. The copy is performed by effectively clearing the target, iterating over the source collection, and inserting each element into the target collection. The iteration is ordered if the source collection is ordered. The target collection semantics are enforced as usual during the insertion process.

`os_list &operator =(const void *e);`

Clears the target collection, inserts the element *e* into the target collection, and returns the target collection.

os_list::operator |=()

`os_list &operator |=(const os_collection &s);`

Inserts the elements contained in *s* into the target collection and returns the target collection.

`os_list &operator |=(const void *e);`

Inserts the element *e* into the target collection and returns the target collection.

os_list::operator &=()

`os_list &operator &=(const os_collection &s);`

For each element in the target collection, reduces the count of the element in the target to the minimum of the counts in the source and target collections. It does so by retaining the appropriate number of leading elements. It returns the target collection.

`os_list &operator &=(const void *e);`

If *e* is present in the target, converts the target into a collection containing just the element *e*. Otherwise, it clears the target collection. It returns the target collection.

os_list::operator –=()

`os_list &operator –=(const os_collection &s);`

For each element in the collection *s*, removes `s.count(e)` occurrences of the element from the target collection. The first `s.count(e)` elements are removed. It returns the target collection.

`os_list &operator –=(const void *e);`

Removes the element *e* from the target collection. The first occurrence of the element is removed from the target collection. It returns the target collection.

os_list::os_list()

`os_list();`

Returns an empty list.

`os_list(os_int32 *expected_size*, os_int32 *behavior* = 0);`

Returns an empty list whose initial implementation is based on the expectation that the *expected_size* argument approximates the usual cardinality of the list, once the list has been loaded with elements.

`os_list(const os_list &*coll*);`

Returns a list that results from assigning the specified list to an empty list.

`os_list(const os_collection &*coll*);`

Returns a list that results from assigning the specified collection to an empty list.

os_nList and os_nlist

```
template <class E>
class os_nList : public os_List<E,
  NUM_PTRS_IN_HEAD,
  NUM_PTRS_IN_BLOCK>

class os_nlist : public os_list<
  NUM_PTRS_IN_HEAD,
  NUM_PTRS_IN_BLOCK>
```

The os_nList and os_nlist classes are parameterized forms of the os_List and os_list classes. Use the os_nList and os_nlist classes to tune the internal list structures for better performance.

If you use os_nList or os_nlist persistently, you must call the macro OS_MARK_NLIST_PT or OS_MARK_NLIST, respectively, in your schema source file for each os_nList or os_nlist that you use. If you are only using os_nList and os_nlist transiently, call the macro OS_TRANSIENT_NLIST in your source file. OS_TRANSIENT_NLIST will define the required stub functions. OS_TRANSIENT_NLIST_NO_BLOCK must be used if you have more than one os_Nlist or os_nlist with the same number of pointers in a block.

For more informations about these macros, see:

- OS_MARK_NLIST() on page 150
- OS_MARK_NLIST_PT() on page 151
- OS_TRANSIENT_NLIST() on page 152
- OS_TRANSIENT_NLIST_NO_BLOCK() on page 153

This section discusses the parameters you can specify for the os_nList and os_nlist classes. For all other API information, see os_List on page 127 and os_list on page 132.

A basic internal structure of an os_list consists of blocks of arrays. Each slot in the array contains a soft pointer to a collection element. When an application creates a list, ObjectStore defines an initial array of slots in the header. The size of this array is determined by NUM_PTRS_IN_HEAD. ObjectStore uses this array to store the initial elements inserted into the collection. When this array becomes full, ObjectStore allocates additional blocks of slots. The size of each block is determined by NUM_PTRS_IN_BLOCKS.

When ObjectStore allocates instances of os_list and os_List, NUM_PTRS_IN_HEAD is set to 4 and NUM_PTRS_IN_BLOCK is set to 8. To perform numerical positional operations on a list, ObjectStore iterates over the blocks to find the block that contains the position. After that, ObjectStore uses simple math to index the correct slot.

To tune a list for faster positional operations, use os_nlist or os_nList. In general, modify NUM_PTRS_IN_BLOCK when you want to tune a collection. Modify NUM_PTRS_IN_HEAD only when you want to tune a very small collection (12 elements or less).

If you create a block or header with too large a number of slots, it has a negative effect during update operations. This can happen when the slots need to be moved up or down, depending on the type of operation.

For example, iteration over blocks for large collections is time consuming. To improve performance, set `NUM_PTRS_IN_BLOCKS` to a number that is larger than 8. This increases the array size for each block. Consequently, there are fewer blocks to iterate over.

Modification of the `NUM_PTRS_IN_HEAD` parameter is more suitable for small collections. You can create the actual list directly in the header, which eliminates additional paging. ObjectStore Technical Support recommends that you never set the `NUM_PTRS_IN_HEAD` parameter to a value larger than 12. A larger value for this parameter defeats its purpose.

Required header file

To use either `os_nList` or `os_nlist`, your application must include the header file:

```
<os_pse/coll/nlist.cc>
```

os_Set

```
template <class E>
class os_Set : public os_Collection<E>
```

A set is an unordered collection that does not allow duplicate element occurrences. The *count* of a value in a given set is the number of times it occurs in the set — either 0 or 1.

The class `os_Set` is *parameterized*, with a parameter for constraining the type of values allowable as elements (for the nonparameterized version of this class, see `os_set` on page 144). The element type parameter, `E`, occurs in the signatures of some of the functions described here. The parameter is used by the compiler to detect type errors.

The element type of any instance of `os_Set` must be a pointer type.

You must mark parameterized collection types in the schema source file.

Tables of member functions and enumerators

The first of the following tables lists the member functions that can be performed on instances of `os_Set`. The second table lists the enumerators inherited by `os_Set` from `os_collection`. Many functions are also inherited by `os_Set` from `os_Collection` or `os_collection`. The full explanation of each inherited function or enumerator appears in the entry for the class from which it is inherited. The full explanation of each function defined by `os_Set` appears in this entry, after the tables. In each case, the *Defined By* column gives the class whose entry contains the full explanation.

Name	*Arguments*	*Returns*	*Defined By*
`cardinality`	`( ) const`	`os_int32`	`os_collection`
`clear`	`( )`	`void`	`os_collection`
`contains`	`( const E ) const`	`os_int32`	`os_Collection`
`count`	`( const E ) const`	`os_int32`	`os_Collection`
`empty`	`( )`	`os_int32`	`os_collection`
`get_behavior`	`( ) const`	`os_unsigned_int32`	`os_collection`
`insert`	`( const E )`	`void`	`os_Collection`
`only`	`( ) const`	`E`	`os_Collection`
`operator os_Array<E>&`	`( )`		`os_Collection`
`operator const os_Array<E>&`	`( ) const`		`os_Collection`
`operator os_array&`	`( )`		`os_collection`
`operator const os_array&`	`( ) const`		`os_collection`
`operator os_Bag<E>&`	`( )`		`os_Collection`

<table>
<tr><th>Name</th><th>Arguments</th><th>Returns</th><th>Defined By</th></tr>
<tr><td>operator const
os_Bag<E>&</td><td>( ) const</td><td></td><td>os_Collection</td></tr>
<tr><td>operator
os_bag&</td><td>( )</td><td></td><td>os_collection</td></tr>
<tr><td>operator const
os_bag&</td><td>( ) const</td><td></td><td>os_collection</td></tr>
<tr><td>operator
os_List<E>&</td><td>( )</td><td></td><td>os_Collection</td></tr>
<tr><td>operator const
os_List<E>&</td><td>( ) const</td><td></td><td>os_Collection</td></tr>
<tr><td>operator
os_list&</td><td>( )</td><td></td><td>os_collection</td></tr>
<tr><td>operator const
os_list&</td><td>( ) const</td><td></td><td>os_collection</td></tr>
<tr><td>operator os_set&</td><td>( )</td><td></td><td>os_collection</td></tr>
<tr><td>operator const
os_set&</td><td>( ) const</td><td></td><td>os_collection</td></tr>
<tr><td>operator ==</td><td>(const os_Collection<E>&) const
(E) const</td><td>os_int32
os_int32</td><td>os_Collection</td></tr>
<tr><td>operator !=</td><td>(const os_Collection<E>&) const
(E) const</td><td>os_int32
os_int32</td><td>os_Collection</td></tr>
<tr><td>operator <</td><td>(const os_Collection<E>&) const
(E) const</td><td>os_int32
os_int32</td><td>os_Collection</td></tr>
<tr><td>operator <=</td><td>(const os_Collection<E>&) const
(E) const</td><td>os_int32
os_int32</td><td>os_Collection</td></tr>
<tr><td>operator ></td><td>(const os_Collection<E>&) const
(E) const</td><td>os_int32
os_int32</td><td>os_Collection</td></tr>
<tr><td>operator >=</td><td>(const os_Collection<E>&) const
(E) const</td><td>os_int32
os_int32</td><td>os_Collection</td></tr>
<tr><td>operator =</td><td>(const os_Set<E>&) const
(const os_Collection<E>&) const
(E) const</td><td>os_Set<E>&
os_Set<E>&
os_Set<E>&</td><td>os_Set</td></tr>
<tr><td>operator |=</td><td>(const os_Collection<E>&) const
(E) const</td><td>os_Set<E>&
os_Set<E>&</td><td>os_Set</td></tr>
<tr><td>operator |</td><td>(const os_Collection<E>&) const
(E) const</td><td>os_Collection<E>&
os_Collection<E>&</td><td>os_Collection</td></tr>
<tr><td>operator &=</td><td>(const os_Collection<E>&) const
(E) const</td><td>os_Set<E>&
os_Set<E>&</td><td>os_Set</td></tr>
<tr><td>operator &</td><td>(const os_Collection<E>&) const
(E) const</td><td>os_Collection<E>&
os_Collection<E>&</td><td>os_Collection</td></tr>
</table>

Name	*Arguments*	*Returns*	*Defined By*
operator -=	(const os_Collection<E>&) const (E) const	os_Set<E>& os_Set<E>&	os_Set
operator -	(const os_Collection<E>&) const (E) const	os_Collection<E>& os_Collection<E>&	os_Collection
os_Set	( ) (os_int32) (const os_Set<E>&) (const os_Collection<E>&)		os_Set
remove	(const E)	os_int32	os_Collection
remove_at	(const os_Cursor<E>&)	void	os_Collection
replace_at	(const E, const os_Cursor<E>&)	E	os_Collection
retrieve	(const os_Cursor<E>&) const	E	os_Collection

os_Set enumerators

The following table lists the enumerators inherited by `os_Set` from `os_collection`.

Name	*Inherited From*
allow_nulls	os_collection
EQ	os_collection
GT	os_collection
LT	os_collection
maintain_order	os_collection
order_by_address	os_collection
unordered	os_collection

os_Set::default_behavior()

```
static os_unsigned_int32 default_behavior();
```

Returns a bit pattern indicating this type’s default behavior.

Assignment Operator Semantics

Assignment operator semantics are described for the following functions in terms of insert operations into the target collection. The actual implementation of the assignment might be different, while still maintaining the associated semantics.

os_Set::operator =()

```
os_Set<E> &operator =(const os_Collection<const E> &s);
```

Copies the contents of the collection *s* into the target collection and returns the target collection. The copy is performed by effectively clearing the target, iterating over the source collection, and inserting each element into the target collection. The iteration is ordered if the source collection is ordered. The target collection semantics are enforced as usual during the insertion process.

```
os_Set<E> &operator =(E e);
```

Clears the target collection, inserts the element *e* into the target collection, and returns the target collection.

os_Set::operator |=()

```
os_Set<E> &operator |=(const os_Collection<E> &s);
```

Inserts the elements contained in *s* into the target collection and returns the target collection.

```
os_Set<E> &operator |=(E e);
```

Inserts the element *e* into the target collection and returns the target collection.

os_Set::operator &=()

```
os_Set<E> &operator &=(const os_Collection<E> &s);
```

For each element in the target collection, reduces the count of the element in the target to the minimum of the counts in the source and target collections. It returns the target collection.

```
os_Set<E> &operator &=(E e);
```

If *e* is present in the target, converts the target into a collection containing just the element *e*. Otherwise, it clears the target collection. It returns the target collection.

os_Set::operator –=()

```
os_Set<E> &operator –=(const os_Collection<E> &s);
```

For each element in the collection *s*, removes `s.count(e)` occurrences of the element from the target collection. It returns the target collection.

```
os_Set<E> &operator –=(E e);
```

Removes the element *e* from the target collection. It returns the target collection.

os_Set::os_Set()

```
os_Set();
```

Returns an empty set.

```
os_Set(os_int32 size);
```

Returns an empty set whose initial implementation is based on the expectation that the *size* argument indicates the approximate usual cardinality of the set, once the set has been loaded with elements.

```
os_Set(const os_Set<E>&);
```

Returns a set that results from assigning the specified set to an empty set.

```
os_Set(const os_Collection<E>&);
```

Returns a set that results from assigning the specified collection to an empty set.

os_set

`class os_set : public os_collection`

A set is an unordered collection that does not allow duplicate element occurrences. The *count* of a value in a given set is the number of times it occurs in the set — either 0 or 1.

The class `os_set` is nonparameterized. For the parameterized version of this class, see `os_Set` on page 139.

Tables of member functions and enumerators

The first of the following tables lists the member functions that can be performed on instances of `os_set`. The second table lists the enumerators inherited by `os_set` from `os_collection`. Many functions are also inherited by `os_set` from `os_collection`. The full explanation of each inherited function or enumerator appears in the entry for the class from which it is inherited. The full explanation of each function defined by `os_set` appears in this entry, after the tables. In each case, the *Defined By* column gives the class whose entry contains the full explanation.

Name	*Arguments*	*Returns*	*Defined By*
cardinality	( ) const	os_int32	os_collection
clear	( )	void	os_collection
contains	(const void*) const		os_collection
count	(const void*) const	os_int32	
default_behavior (static)	( )	os_unsigned_int32	os_set
empty	( )	os_int32	os_collection
get_behavior	( ) const	os_unsigned_int32	os_collection
insert	(const void*)	void	os_collection
only	( ) const	void*	os_collection
operator os_array&	( )		os_collection
operator const os_array&	( ) const		os_collection
operator os_bag&	( )		os_collection
operator const os_bag&	( ) const		os_collection
operator os_ list&	( )		os_collection
operator const os_list&	( ) const		os_collection
operator ==	(const os_collection&) const (const void*) const	os_int32 os_int32	os_collection
operator !=	(const os_collection&) const (const void*) const	os_int32 os_int32	os_collection

<table>
<tr><th>Name</th><th>Arguments</th><th>Returns</th><th>Defined By</th></tr>
<tr><td>operator <</td><td>(const os_collection&) const
(const void*) const</td><td>os_int32
os_int32</td><td>os_collection</td></tr>
<tr><td>operator <=</td><td>(const os_collection&) const
(const void*) const</td><td>os_int32
os_int32</td><td>os_collection</td></tr>
<tr><td>operator ></td><td>(const os_collection&) const
(const void*) const</td><td>os_int32
os_int32</td><td>os_collection</td></tr>
<tr><td>operator >=</td><td>(const os_collection&) const
(const void*) const</td><td>os_int32
os_int32</td><td>os_collection</td></tr>
<tr><td>operator =</td><td>(const os_set&) const
(const os_collection&) const
(const void*) const</td><td>os_set&
os_set&
os_set</td><td>os_set</td></tr>
<tr><td>operator |=</td><td>(const os_collection&) const
(const void*) const</td><td>os_set&
os_set&</td><td>os_set</td></tr>
<tr><td>operator |</td><td>(const os_collection&) const
(const void*) const</td><td>os_set&
os_set&</td><td>os_collection</td></tr>
<tr><td>operator &=</td><td>(const os_collection&) const
(const void*) const</td><td>os_set&
os_set&</td><td>os_set</td></tr>
<tr><td>operator &</td><td>(const os_collection&) const
(const void*) const</td><td>os_set&
os_set&</td><td>os_collection</td></tr>
<tr><td>operator -=</td><td>(const os_collection&) const
(const void*) const</td><td>os_set&
os_set&</td><td>os_set</td></tr>
<tr><td>operator -</td><td>(const os_collection&) const
(const void*) const</td><td>os_set&
os_set&</td><td>os_collection</td></tr>
<tr><td>os_set</td><td>( )
(os_int32)
(const os_set&)
(const os_collection&)</td><td></td><td>os_set</td></tr>
<tr><td>remove</td><td>(const void*)</td><td>os_int32</td><td>os_collection</td></tr>
<tr><td>remove_at</td><td>(const os_cursor&)</td><td>void</td><td>os_collection</td></tr>
<tr><td>replace_at</td><td>(const void*,
const os_cursor&)</td><td>void*</td><td>os_collection</td></tr>
<tr><td>retrieve</td><td>(const os_cursor&) const</td><td>void*</td><td>os_collection</td></tr>
</table>

os_set enumerators

The following table lists the enumerators inherited by `os_set` from `os_collection`.

Name	*Inherited From*
`allow_duplicates`	`os_collection`
`allow_nulls`	`os_collection`
`EQ`	`os_collection`
`GT`	`os_collection`
`LT`	`os_collection`
`maintain_order`	`os_collection`

Assignment Operator Semantics

Assignment operator semantics are described for the following functions in terms of insert operations into the target collection. The actual implementation of the assignment might be different, while still maintaining the associated semantics.

os_set::operator =()

`os_set &operator =(const os_set &s);`

Copies the contents of the collection *s* into the target collection and returns the target collection. The copy is performed by effectively clearing the target, iterating over the source collection, and inserting each element into the target collection. The iteration is ordered if the source collection is ordered. The target collection semantics are enforced as usual during the insertion process.

`os_set &operator =(const void *e);`

Clears the target collection, inserts the element *e* into the target collection, and returns the target collection.

os_set::operator |=()

`os_set &operator |=(const os_set &s);`

Inserts the elements contained in *s* into the target collection and returns the target collection.

`os_set &operator |=(const void *e);`

Inserts the element *e* into the target collection and returns the target collection.

os_set::operator &=()

```
os_set &operator &=(const os_set &s);
```

For each element in the target collection, reduces the count of the element in the target to the minimum of the counts in the source and target collections. If the collection is ordered and contains duplicates, it does so by retaining the appropriate number of leading elements. It returns the target collection.

```
os_set &operator &=(const void *e);
```

If `e` is present in the target, converts the target into a collection containing just the element `e`. Otherwise, it clears the target collection. It returns the target collection.

os_set::operator –=()

```
os_set &operator –=(const os_set &s);
```

For each element in the collection `s`, removes `s.count(e)` occurrences of the element from the target collection. If the collection is ordered, it is the first `s.count(e)` elements that are removed. It returns the target collection.

```
os_set &operator –=(const void *e);
```

Removes the element `e` from the target collection. If the collection is ordered, it is the first occurrence of the element that is removed from the target collection. It returns the target collection.

os_set::os_set()

```
os_set();
```

Returns an empty set.

```
os_set(os_int32 size);
```

Returns an empty set whose initial implementation is based on the expectation that the `size` argument indicates the approximate usual cardinality of the set, once the set has been loaded with elements.

```
os_set(const os_set&);
```

Returns a set that results from assigning the specified set to an empty set.

```
os_set(const os_collection&);
```

Returns a set that results from assigning the specified collection to an empty set.

Chapter 7
Macros and User-Defined Functions Reference

OS_MARK_DICTIONARY()

If you use persistent dictionaries, or any combination of persistent and transient dictionaries, you must call the macro `OS_MARK_DICTIONARY()` for each key-type/element-type pair that you use.

Form of the call

`OS_MARK_DICTIONARY(`*`key_type`*`, `*`element_type`*`)`

Put these calls in the same function with your calls to `OS_MARK_SCHEMA_TYPE()`. For example:

```
/*** schema.cc ***/

#include <ostore/ostore.hh>
#include <ostore/coll.hh>
#include <ostore/coll/dict_pt.hh>
#include <ostore/manschem.hh>
#include "dnary.hh"

  OS_MARK_DICTIONARY(void*,Course*);
  OS_MARK_DICTIONARY(int,Employee**);
  OS_MARK_SCHEMA_TYPE(Course);
  OS_MARK_SCHEMA_TYPE(Employee);
  OS_MARK_SCHEMA_TYPE(Department);
```

For pointer keys, specify `void*` as the *`key_type`*.

For class keys, the class must have a destructor, and you must register rank and hash functions for the class.

If you use transient dictionaries, you must call the macro `OS_TRANSIENT_DICTIONARY()`. The arguments are the same as for `OS_MARK_DICTIONARY()`, but you call `OS_TRANSIENT_DICTIONARY()` at file scope in an application source file, rather than at function scope in a schema source file.

OS_MARK_NLIST()

If you use a persistent `os_nlist`, you must call the macro `OS_MARK_NLIST` for each set of `os_nlist` parameters that you use.

Form of the call

`OS_MARK_NLIST(`*`num_ptrs_in_head`*`,`*`num_ptrs_in_block`*`)`

Put these calls in the same function with your calls to `OS_MARK_SCHEMA_TYPE()`. You must include the header file `<os_pse/coll/nlist.hh>`. For example:

```
/*** schema.cc ***/
OS_MARK_NLIST(8,16);
OS_MARK_NLIST(4,20);
```

If you use `os_nlist` transiently, you must call the macro `OS_TRANSIENT_NLIST()`. If you use a combination of persistent and transient `os_nlists`, it is sufficient to just call `OS_MARK_NLIST()` for each unique set of template parameters.

OS_MARK_NLIST_PT()

If you use a persistent `os_nList`, you must call the macro `OS_MARK_NLIST_PT` for each set of `os_nList` parameters that you use.

Form of the call

`OS_MARK_NLIST_PT(`*`E`*`,`*`num_ptrs_in_head`*`,`*`num_ptrs_in_block`*`)`

Put these calls in the same function with your calls to `OS_MARK_SCHEMA_TYPE()`. You must include the header file `<os_pse/coll/nlist.hh>`. For example:

```
/*** schema.cc ***/
OS_MARK_NLIST_PT(Object*,8,16);
OS_MARK_NLIST_PT(Object*,4,20);
OS_MARK_NLIST_PT(X*,4,20);
```

If you use `os_nList` transiently, you must call the macro `OS_TRANSIENT_NLIST()`. If you use a combination of persistent and transient `os_nLists`, it is sufficient to just call `OS_MARK_NLIST_PT()` for each unique set of template parameters.

OS_TRANSIENT_DICTIONARY()

If you use only transient dictionaries, you must call the macro `OS_TRANSIENT_DICTIONARY()` for each key-type/element-type pair that you use. This is true unless there are ObjectStore dictionaries with the same key marked persistently. In this case, the macro is not needed and its use produces error messages at link time.

Form of the call

`OS_TRANSIENT_DICTIONARY(`*`key_type`*`, `*`element_type`*`)`

Here are some examples:

```
OS_TRANSIENT_DICTIONARY(void*,Course*);
OS_TRANSIENT_DICTIONARY(int,Employee**);
```

Put these calls at file scope in an application source file.

For pointer keys, specify `void*` as the *`key_type`*.

For class keys, the class must have an `operator=` and a destructor that zeroes out any pointers in the key object.

If a transient `os_Dictionary` is instantiated and `OS_TRANSIENT_DICTIONARY()` is missing, `_Rhash_pt<KEYTYPE>::get_os_typespec()` and `_Dict_pt_slot<KEYTYPE>::get_os_typespec()` are undefined at link time.

Using user-defined classes

In order to use a user-defined class as a key, you must have `get_os_typespec()` declared and defined as follows, where `KEYTYPE` is the name of the user-defined class:

```
{ return new os_typespec("KEYTYPE"); }
```

OS_TRANSIENT_DICTIONARY_NOKEY()

If you use only transient dictionaries, you must call the macro `OS_TRANSIENT_DICTIONARY_NOKEY()` in certain cases where you have more than one dictionary defined with the same key type.

Form of the call

`OS_TRANSIENT_DICTIONARY_NOKEY(`*`key_type`*`)`

`OS_TRANSIENT_DICTIONARY()` defines stubs for `get_os_typespec()` member functions of internal data structures parameterized by either the key type and the value type, or by just the key type. If you have in your application more than one dictionary with the same key type, specifying `OS_TRANSIENT_DICTIONARY()` multiple times will result in multiply-defined symbols at link time. Instead, use `OS_TRANSIENT_DICTIONARY_NOKEY()`, which defines just the `get_os_typespec()` functions for internal data structures parameterized by both the key and value type.

For example, if you had

```
os_Dictionary<int, Object1*> d1;
```

```
os_Dictionary<int, Object2*> d2;
```

You would use

```
OS_TRANSIENT_DICTIONARY(int, Object1*);
OS_TRANSIENT_DICTIONARY_NOKEY(int, Object2*);
```

Put these calls at file scope in an application source file.

For pointer keys, specify `void*` as *`key_type`*.

For class keys, the class must have a destructor.

The user-defined class that is used as a key must have `get_os_typespec()` declared and defined. `get_os_typespec()` should be defined as follows, where `KEYTYPE` is the name of the user-defined class:

```
{ return new os_typespec("KEYTYPE"); }
```

OS_TRANSIENT_NLIST()

If you use only transient `os_nlist` and `os_nList`, you must call the macro `OS_TRANSIENT_NLIST()` for each unique set of template parameters given to your `os_nlist` or `os_nList`. This is true unless there is a persistent `os_nlist` or `os_nList` with the same template parameters and for which you have called `OS_MARK_NLIST` or `OS_MARK_NLIST_PT` in your schema source file. In this case, the macro is not needed and will produce an error message at link time.

Form of the call

`OS_TRANSIENT_NLIST(`*`num_ptrs_in_head`*`,`*`num_ptrs_in_block`*`)`

Here are some examples:

```
os_nlist(8,16) nl1;
os_nList(Object*,8,20) nl2;
...
```

```
OS_TRANSIENT_NLIST(8,16)
OS_TRANSIENT_NLIST(8,20)
```

Put the calls to `OS_TRANSIENT_NLIST` at file scope in an application source file.

If a transient `os_nlist` or `os_nList` is instantiated and `OS_TRANSIENT_NLIST()` is missing, `os_nlist<`*`num_ptrs_in_head`*`,`*`num_ptrs_in_block`*`>::get_os_typespec()` and `os_chained_list_block_pt<`*`num_ptrs_in_block`*`>::get_os_typespec()` are undefined at link time.

OS_TRANSIENT_NLIST_NO_BLOCK()

If you use a transient `os_nlist` or `os_nList`, you must call the macro `OS_TRANSIENT_NLIST_NO_BLOCK` in certain cases where you have more than one `os_nlist` or `os_nList` with the same *`num_ptrs_in_block`* template parameter.

Form of the call

`OS_TRANSIENT_NLIST_NO_BLOCK(`*`num_ptrs_in_head`*`)`

`OS_TRANSIENT_NLIST` defines stubs for `get_os_typespec()` member functions of internal data structures parameterized by either the *`num_ptrs_in_head`* and *`num_ptrs_in_block`* arguments, or by just the *`num_ptrs_in_block`* argument. If you have in your application more than one `os_nlist` or `os_nList` with the same *`num_ptrs_in_block`* argument, specifying `OS_TRANSIENT_NLIST` multiple times will result in multiply defined symbols at link time. Instead, use `OS_TRANSIENT_NLIST` for one set of parameters, and use `OS_TRANSIENT_NLIST_NO_BLOCK` for any others with the same value for *`num_ptrs_in_block`*.

Here are some examples:

```
os_nlist(8,16) nl1;
os_nlist(8,16) nl2;
os_nlist(4,16) nl3;
...
OS_TRANSIENT_NLIST(8,16)
OS_TRANSIENT_NLIST_NO_BLOCK(4)
```

Put the calls to `OS_TRANSIENT_NLIST_NO_BLOCK` at file scope in an application source file.

os_assign_function()

This macro is used to register an assignment function for an ordered os_Dictionary key class or for an ordered or unordered index key class. Use this macro to make it possible to use a key class that contains a soft pointer or to use an assignment operator instead of memcpy when inserting or reorganizing objects in the dictionary or index.

Use this macro in conjunction with the macro os_assign_function_body() on page 154. This function must be called after os_collection::initialize().

To use the collections facility, you must include the file <os_pse/coll.hh> after including <os_pse/ostore.hh>.

Form of the call

os_assign_function(*class*)

class is the class used as a key.

The os_assign_function() macro should be invoked before creating the ordered os_Dictionary or adding the index.

os_assign_function_body()

This macro is used to create an assignment function body for an ordered os_Dictionary key class or for an ordered or unordered index key class. The function body consists of an operator= statement. This macro makes it possible to use a key class that contains a soft pointer or to use an assignment operator instead of memcpy when inserting or reorganizing objects in the dictionary or index.

Use this macro in conjunction with the macro os_assign_function() on page 154.

To use the collections facility, you must include the file <os_pse/coll.hh> after including <os_pse/ostore.hh>.

Form of the call

os_assign_function_body(*class*)

class is the class used as a key.

The os_assign_function_body() macro should be invoked at module level in a source file (for example, the file containing the definition of the member function).

os_index_key()

This macro is used to register user-defined rank and hash functions with PSE Pro.

To use the collections facility, you must include the file `<os_pse/coll.hh>` after including `<os_pse/ostore.hh>`. This function must be called after `os_collection::initialize()`.

Form of the call

`os_index_key(`*`class`*`,`*`rank_function`*`,`*`hash_function`*`)`

`class` is the class whose instances are ranked or hashed by the specified functions.

`rank_function` is a user-defined function that, for any pair of instances of *`class`*, provides an ordering indicator for the instances, much as `strcmp()` does for arrays of characters. You must supply this function. The rank function should return one of `os_collection::LT`, `os_collection::GT`, or `os_collection::EQ`. See Supplying Rank and Hash Functions on page 58.

`hash_function` is a user-defined function that, for each instance of *`class`*, returns an `os_unsigned_int32` value that can be used as a key in a hash table. Supplying this function is optional. If you do not supply a hash function for the class, specify `0` as the hash function argument.

Caution

The macro arguments are used (among other things) to concatenate unique names. The details of macro preprocessing differ from compiler to compiler, and in some cases it is necessary to enter these macro arguments *without white space* to ensure that the argument concatenation will work correctly.

os_index_key_hash_function()

This macro is used to register user-defined hash functions with ObjectStore. Use it only to *replace* a hash function registered previously.

To use the collections facility, you must include the file `<os_pse/coll.hh>` after including `<os_pse/ostore.hh>`. This function must be called after `os_collection::initialize()`.

Form of the call

`os_index_key_hash_function(`*`class`*`,`*`hash_function`*`)`

`class` is the class whose instances are hashed by the specified function.

`hash_function` is a user-defined function that, for each instance of *`class`*, returns an `os_unsigned_int32` value that can be used as a key in a hash table.

Caution

The macro arguments are used (among other things) to concatenate unique names. The details of macro preprocessing differ from compiler to compiler, and in some cases it is necessary to enter these macro arguments *without white space* to ensure that the argument concatenation will work correctly.

os_index_key_rank_function()

This macro is used to register user-defined rank functions with ObjectStore. Use it only to *replace* a rank function registered previously.

To use the collections facility, you must include the file `<os_pse/coll.hh>` after including `<os_pse/ostore.hh>`. This function must be called after `os_collection::initialize()`.

Form of the call

`os_index_key_rank_function(`*`class`*`,`*`rank_function`*`)`

`class` is the class whose instances are ranked by the specified function. *`class`* can also be `char*` when you register `os_strcoll_for_char_pointer()`, and `char[]` when you register `os_strcoll_for_char_array()`. These versions of `strcoll()`, provided by ObjectStore, will be used, if registered, instead of `strcmp()` to support indexes keyed by `char*` or `char[]`.

`rank_function` is a user-defined function that, for any pair of instances of *`class`*, provides an ordering indicator for the instances, much as `strcmp()` does for arrays of characters. The rank function should return one of `os_collection::LT`, `os_collection::GT`, or `os_collection::EQ`. See Supplying Rank and Hash Functions on page 58.

Caution

The macro arguments are used (among other things) to concatenate unique names. The details of macro preprocessing differ from compiler to compiler, and in some cases it is necessary to enter these macro arguments *without white space* to ensure that the argument concatenation will work correctly.

OS_INDEXABLE_LINKAGE()

Note

This macro is for use on Windows platforms only.

Specifies the linkage for classes generated by the `os_indexable_`*`xxx`* macros. This macro can be used with component schemas on Windows platforms. For example, you could define the macro as Microsoft's `__declspec (dllexport)` to allow one DLL to create a subclass of a class defined in another DLL when there are relationship members.

You must define `OS_INDEXABLE_LINKAGE()` before including `<ostore/coll.hh>`. For example:

```
...
#define OS_INDEXABLE_LINKAGE __declspec(dllexport)
#include <ostore/coll.hh>
```

If the macro is not defined, the default is blank.

os_query_function_body_with_namespace()

Applications that use a member function in a query or path string, where the function is a member of a class encapsulated within a namespace, must call this macro.

Form of the call

```
os_query_function_body_body_with_namespace(
    class,qualified_class,return_type,bpname)
```

`class` is the name of the class that defines the member function

`qualified_class` is the name of the class, qualified with its namespace, for example. `NameSpace::class_name`, that defines the member function

`return_type` names the type of value returned by the member function

`bpname` is the name of the `os_backptr`-valued member of class

The `os_query_function_body_with_namespace()` macro should be invoked at module level in a source file (for example, the file containing the definition of the member function). No white space should appear in the argument list.

os_rel_1_1_body()

ObjectStore allows the user to model binary relationships with pointer-valued (or collection-of-pointer-valued) data members that maintain the referential integrity of their inverse data members. You implement this inverse maintenance by defining an embedded relationship class that encapsulates the pointer (or collection-of-pointer) so that it can intercept updates to the encapsulated value and perform the necessary inverse maintenance tasks. The encapsulated-pointer values are stored as soft pointers so as to maintain the values across address space release and transactions.

Required include files

To use this macro, you must include the file <os_pse/relat.hh> after including <os_pse/ostore.hh>. If you also include <os_pse/coll.hh>, include <os_pse/relat.hh> after both <os_pse/ostore.hh> and <os_pse/coll.hh>.

The actual value type of a data member with an inverse is a special class whose instances encapsulate the member's apparent value. This implicitly defined class defines `operator =()` (for setting the apparent value), as well as `operator ->()`, `operator *()`, and a conversion operator for converting its instances to instances of the apparent value type (for getting the apparent value). Under most circumstances, these operators make the encapsulating objects transparent.

The implicitly defined class also defines the member functions `getvalue()`, which returns the apparent value, and `setvalue()`, which takes an instance of the apparent value type as argument. These functions can always be used to set and get the member's apparent value explicitly.

This macro is used to instantiate accessor functions for a single-valued data member with a single-valued inverse data member. Calls to this macro should appear at the top level in a source file associated with the class defining the member.

Form of the call

`os_rel_1_1_body(`*`class`*`,`*`member`*`,`*`inv_class`*`,`*`inv_mem`*`)`

`class` is the class defining the data member being declared.

`member` is the name of the member being declared.

`inv_class` is the name of the class that defines the inverse member.

`inv_mem` is the name of the inverse member.

Caution

The macro arguments are used (among other things) to concatenate unique names for the encapsulating relationship class and its accessor functions. The details of macro preprocessing differ from compiler to compiler, and in some cases it is necessary to enter these macro arguments *without white space* to ensure that the argument concatenation will work correctly.

os_rel_1_m_body()

ObjectStore allows the user to model binary relationships with pointer-valued (or collection-of-pointer-valued) data members that maintain the referential integrity of their inverse data members. You implement this inverse maintenance by defining an embedded relationship class, which encapsulates the pointer (or collection-of-pointer) so that it can intercept updates to the encapsulated value and perform the necessary inverse maintenance tasks. The encapsulated-pointer values are stored as soft pointers so as to maintain the values across address space release and transactions.

Required include files

To use this macro, you must include the file `<os_pse/relat.hh>` after including `<os_pse/ostore.hh>` and `<os_pse/coll.hh>`.

The actual value type of a data member with an inverse is a special class whose instances encapsulate the member's apparent value. This implicitly defined class defines `operator =()` (for setting the apparent value), as well as `operator ->()`, `operator *()`, and a conversion operator for converting its instances to instances of the apparent value type (for getting the apparent value). Under most circumstances, these operators make the encapsulating objects transparent.

The implicitly defined class also defines the member functions `getvalue()`, which returns the apparent value, and `setvalue()`, which takes an instance of the apparent value type as argument. These functions can always be used to set and get the member's apparent value explicitly.

This macro is used to instantiate accessor functions for a single-valued data member with a many-valued inverse data member. Calls to this macro should appear at the top level in the source file associated with the class defining the member.

Form of the call

`os_rel_1_m_body(`*`class`*`,`*`member`*`,`*`inv_class`*`,`*`inv_mem`*`)`

`class` is the class defining the data member being declared.

`member` is the name of the member being declared.

`inv_class` is the name of the class that defines the inverse member.

`inv_mem` is the name of the inverse member.

Caution

The macro arguments are used (among other things) to concatenate unique names for the embedded relationship class and its accessor functions. The details of macro preprocessing differ from compiler to compiler, and in some cases it is necessary to enter these macro arguments *without white space* to ensure that the argument concatenation will work correctly.

os_rel_m_1_body()

ObjectStore allows the user to model binary relationships with pointer-valued (or collection-of-pointer-valued) data members that maintain the referential integrity of their inverse data members. You implement this inverse maintenance by defining an embedded relationship class, which encapsulates the pointer (or collection-of-pointer) so that it can intercept updates to the encapsulated value and perform the necessary inverse maintenance tasks. The encapsulated-pointer values are stored as soft pointers so as to maintain the values across address space release and transactions.

Required include files

To use this macro, you must include the file <os_pse/relat.hh> after including <os_pse/ostore.hh> and <os_pse/coll.hh>.

The actual value type of a data member with an inverse is a special class whose instances encapsulate the member's apparent value. This implicitly defined class defines `operator =()` (for setting the apparent value), as well as `operator ->()`, `operator *()`, and a conversion operator for converting its instances to instances of the apparent value type (for getting the apparent value). Under most circumstances, these operators make the encapsulating objects transparent.

The implicitly defined class also defines the member functions `getvalue()`, which returns the apparent value, and `setvalue()`, which takes an instance of the apparent value type as argument. These functions can always be used to set and get the member's apparent value explicitly.

This macro is used to instantiate accessor functions for a many-valued data member with a single-valued inverse data member. Calls to this macro should appear at the top level in the source file associated with the class defining the member.

Form of the call

`os_rel_m_1_body(`*`class`*`,`*`member`*`,`*`inv_class`*`,`*`inv_mem`*`)`

`class` is the class defining the data member being declared.

`member` is the name of the member being declared.

`inv_class` is the name of the class that defines the inverse member.

`inv_mem` is the name of the inverse member.

Caution

The macro arguments are used (among other things) to concatenate unique names for the embedded relationship class and its accessor functions. The details of macro preprocessing differ from compiler to compiler, and in some cases it is necessary to enter these macro arguments *without white space* to ensure that the argument concatenation will work correctly.

os_rel_m_m_body()

ObjectStore allows the user to model binary relationships with pointer-valued (or collection-of-pointer-valued) data members that maintain the referential integrity of their inverse data members. You implement this inverse maintenance by defining an embedded relationship class, which encapsulates the pointer (or collection-of-pointer) so that it can intercept updates to the encapsulated value and perform the necessary inverse maintenance tasks. The encapsulated-pointer values are stored as soft pointers so as to maintain the values across address space release and transactions.

Required include files

To use this macro, you must include the file <os_pse/relat.hh> after including <os_pse/ostore.hh> and <os_pse/coll.hh>.

The actual value type of a data member with an inverse is a special class whose instances encapsulate the member's apparent value. This implicitly defined class defines `operator =()` (for setting the apparent value), as well as `operator ->()`, `operator *()`, and a conversion operator for converting its instances to instances of the apparent value type (for getting the apparent value). Under most circumstances, these operators make the encapsulating objects transparent.

The implicitly defined class also defines the member functions `getvalue()`, which returns the apparent value, and `setvalue()`, which takes an instance of the apparent value type as argument. These functions can always be used to set and get the member's apparent value explicitly.

This macro is used to instantiate accessor functions for a many-valued data member with a many-valued inverse data member. Calls to this macro should appear at the top level in the source file associated with the class defining the member.

Form of the call

`os_rel_m_m_body(`*`class`*`,`*`member`*`,`*`inv_class`*`,`*`inv_mem`*`)`

`class` is the class defining the data member being declared.

`member` is the name of the member being declared.

`inv_class` is the name of the class that defines the inverse member.

`inv_mem` is the name of the inverse member.

Caution

The macro arguments are used (among other things) to concatenate unique names for the encapsulating relationship class and its accessor functions. The details of macro preprocessing differ from compiler to compiler, and in some cases it is necessary to enter these macro arguments *without white space* to ensure that the argument concatenation will work correctly.

os_rel_1_1_body_options()

ObjectStore allows the user to model binary relationships with pointer-valued (or collection-of-pointer-valued) data members that maintain the referential integrity of their inverse data members. You implement this inverse maintenance by defining an embedded relationship class, which encapsulates the pointer (or collection-of-pointer) so that it can intercept updates to the encapsulated value and perform the necessary inverse maintenance tasks. The encapsulated-pointer values are stored as soft pointers so as to maintain the values across address space release and transactions.

Required include files

To use this macro, you must include the file <os_pse/relat.hh> after including <os_pse/ostore.hh>. If you also include <os_pse/coll.hh>, include <os_pse/relat.hh> after both <os_pse/ostore.hh> and <os_pse/coll.hh>.

The actual value type of a data member with an inverse is a special class whose instances encapsulate the member's apparent value. This implicitly defined class defines `operator =()` (for setting the apparent value), as well as `operator ->()`, `operator *()`, and a conversion operator for converting its instances to instances of the apparent value type (for getting the apparent value). Under most circumstances, these operators make the encapsulating objects transparent.

The implicitly defined class also defines the member functions `getvalue()`, which returns the apparent value, and `setvalue()`, which takes an instance of the apparent value type as argument. These functions can always be used to set and get the member's apparent value explicitly.

This macro is used to instantiate accessor functions for a single-valued data member with a single-valued inverse data member, when deletion propagation is desired. Calls to this macro should appear at the top level in the source file associated with the class defining the member.

Form of the call

```
os_rel_1_1_body_options(class,member,inv_class,inv_mem,
  deletion, index, inv_index)
```

class is the class defining the data member being declared.

member is the name of the member being declared.

inv_class is the name of the class that defines the inverse member.

inv_mem is the name of the inverse member.

deletion is either `os_rel_propagate_delete` or `os_rel_dont_propagate_delete`. By default, deleting an object that participates in a relationship automatically updates the other side of the relationship so that there are no dangling pointers to the deleted object. In some cases, however, the desired behavior is actually to delete the object on the other side of the relationship (for example, for subsidiary component objects). This behavior is specified with `os_rel_propagate_delete`.

index specifies whether the current member is indexable. For nonindexable members, use `os_no_index`. PSE Pro does not support indexable members.

inv_index specifies whether the inverse member is indexable. For nonindexable members, use `os_no_index`. PSE Pro does not support indexable members.

Caution

The first four macro arguments are used (among other things) to concatenate unique names for the encapsulating relationship class and its accessor functions. The details of macro preprocessing differ from compiler to compiler, and in some cases it is necessary to enter these macro arguments *without white space* to ensure that the argument concatenation will work correctly. There should be no white space in the argument list between the opening parenthesis and the comma separating the fourth and fifth arguments.

os_rel_1_m_body_options()

ObjectStore allows the user to model binary relationships with pointer-valued (or collection-of-pointer-valued) data members that maintain the referential integrity of their inverse data members. You implement this inverse maintenance by defining an embedded relationship class, which encapsulates the pointer (or collection-of-pointer) so that it can intercept updates to the encapsulated value and perform the necessary inverse maintenance tasks. The encapsulated-pointer values are stored as soft pointers so as to maintain the values across address space release and transactions.

Required include files

To use this macro, you must include the file <os_pse/relat.hh> after including <os_pse/ostore.hh> and <os_pse/coll.hh>.

The actual value type of a data member with an inverse is a special class whose instances encapsulate the member's apparent value. This implicitly defined class defines `operator =()` (for setting the apparent value), as well as `operator ->()`, `operator *()`, and a conversion operator for converting its instances to instances of the apparent value type (for getting the apparent value). Under most circumstances, these operators make the encapsulating objects transparent.

The implicitly defined class also defines the member functions `getvalue()`, which returns the apparent value, and `setvalue()`, which takes an instance of the apparent value type as argument. These functions can always be used to set and get the member's apparent value explicitly.

This macro is used to instantiate accessor functions for a single-valued data member with a many-valued inverse data member, when deletion propagation is desired. Calls to this macro should appear at the top level in the source file associated with the class defining the member.

Form of the call

```
os_rel_1_m_body_options(class,member,inv_class,inv_mem,
  deletion, index, inv_index)
```

class is the class defining the data member being declared.

member is the name of the member being declared.

inv_class is the name of the class that defines the inverse member.

inv_mem is the name of the inverse member.

deletion is either `os_rel_propagate_delete` or `os_rel_dont_propagate_delete`. By default, deleting an object that participates in a relationship automatically updates the other side of the relationship so that there are no dangling pointers to the deleted object. In some cases, however, the desired behavior is actually to delete the object on the other side of the relationship (for example, for subsidiary component objects). This behavior is specified with `os_rel_propagate_delete`.

index specifies whether the current member is indexable. For nonindexable members, use `os_no_index`. PSE Pro does not support indexable members.

inv_index specifies whether the inverse member is indexable. For nonindexable members, use `os_no_index`. PSE Pro does not support indexable members.

Caution

The first four macro arguments are used (among other things) to concatenate unique names for the encapsulating relationship class and its accessor functions. The details of macro preprocessing differ from compiler to compiler, and in some cases it is necessary to enter these macro arguments *without white space* to ensure that the argument concatenation will work correctly. There should be no white space in the argument list between the opening parenthesis and the comma separating the fourth and fifth arguments.

os_rel_m_1_body_options()

ObjectStore allows the user to model binary relationships with pointer-valued (or collection-of-pointer-valued) data members that maintain the referential integrity of their inverse data members. You implement this inverse maintenance by defining an embedded relationship class, which encapsulates the pointer (or collection-of-pointer) so that it can intercept updates to the encapsulated value and perform the necessary inverse maintenance tasks. The encapsulated-pointer values are stored as soft pointers so as to maintain the values across address space release and transactions.

Required include files

To use this macro, you must include the file <os_pse/relat.hh> after including <os_pse/ostore.hh> and <os_pse/coll.hh>.

The actual value type of a data member with an inverse is a special class whose instances encapsulate the member's apparent value. This implicitly defined class defines `operator =()` (for setting the apparent value), as well as `operator ->()`, `operator *()`, and a conversion operator for converting its instances to instances of the apparent value type (for getting the apparent value). Under most circumstances, these operators make the encapsulating objects transparent.

The implicitly defined class also defines the member functions `getvalue()`, which returns the apparent value, and `setvalue()`, which takes an instance of the apparent value type as argument. These functions can always be used to set and get the member's apparent value explicitly.

This macro is used to instantiate accessor functions for a many-valued data member with a single-valued inverse data member, when deletion propagation is desired.

Calls to this macro should appear at the top level in the source file associated with the class defining the member.

Form of the call

```
os_rel_m_1_body_options(class,member,inv_class,inv_mem,
  deletion, index, inv_index)
```

class is the class defining the data member being declared.

member is the name of the member being declared.

inv_class is the name of the class that defines the inverse member.

inv_mem is the name of the inverse member.

deletion is either `os_rel_propagate_delete` or `os_rel_dont_propagate_delete`. By default, deleting an object that participates in a relationship automatically updates the other side of the relationship so that there are no dangling pointers to the deleted object. In some cases, however, the desired behavior is actually to delete the object on the other side of the relationship (for example, for subsidiary component objects). This behavior is specified with `os_rel_propagate_delete`.

index specifies whether the current member is indexable. For nonindexable members, use `os_no_index`.

inv_index specifies whether the inverse member is indexable. For nonindexable members, use `os_no_index`. PSE Pro does not support indexable members.

Caution

The first four macro arguments are used (among other things) to concatenate unique names for the encapsulating relationship class and its accessor functions. The details of macro preprocessing differ from compiler to compiler, and in some cases it is necessary to enter these macro arguments *without white space* to ensure that the argument concatenation will work correctly. There should be no white space in the argument list between the opening parenthesis and the comma separating the fourth and fifth arguments.

os_rel_m_m_body_options()

ObjectStore allows the user to model binary relationships with pointer-valued (or collection-of-pointer-valued) data members that maintain the referential integrity of their inverse data members. You implement this inverse maintenance by defining an embedded relationship class, which encapsulates the pointer (or collection-of-pointer) so that it can intercept updates to the encapsulated value and perform the necessary inverse maintenance tasks. The encapsulated-pointer values are stored as soft pointers so as to maintain the values across address space release and transactions.

Required include files

To use this macro, you must include the file <os_pse/relat.hh> after including <os_pse/ostore.hh> and <os_pse/coll.hh>.

The actual value type of a data member with an inverse is a special class whose instances encapsulate the member's apparent value. This implicitly defined class defines `operator =()` (for setting the apparent value), as well as `operator ->()`, `operator *()`, and a conversion operator for converting its instances to instances of the apparent value type (for getting the apparent value). Under most circumstances, these operators make the encapsulating objects transparent.

The implicitly defined class also defines the member functions `getvalue()`, which returns the apparent value, and `setvalue()`, which takes an instance of the apparent value type as argument. These functions can always be used to set and get the member's apparent value explicitly.

This macro is used to instantiate accessor functions for a many-valued data member with a many-valued inverse data member, when deletion propagation is desired. Calls to this macro should appear at the top level in the source file associated with the class defining the member.

Form of the call

```
os_rel_m_m_body_options(class,member,inv_class,inv_mem,
  deletion, index, inv_index)
```

class is the class defining the data member being declared.

member is the name of the member being declared.

inv_class is the name of the class that defines the inverse member.

inv_mem is the name of the inverse member.

deletion is either `os_rel_propagate_delete` or `os_rel_dont_propagate_delete`. By default, deleting an object that participates in a relationship automatically updates the other side of the relationship so that there are no dangling pointers to the deleted object. In some cases, however, the desired behavior is actually to delete the object on the other side of the relationship (for example, for subsidiary component objects). This behavior is specified with `os_rel_propagate_delete`.

index specifies whether the current member is indexable. For nonindexable members, use `os_no_index`. PSE Pro does not support indexable members.

inv_index specifies whether the inverse member is indexable. For nonindexable members, use `os_no_index`. PSE Pro does not support indexable members.

Caution

The first four macro arguments are used (among other things) to concatenate unique names for the encapsulating relationship class and its accessor functions. The details of macro preprocessing differ from compiler to compiler, and in some cases it is necessary to enter these macro arguments *without white space* to ensure that the argument concatenation will work correctly. There should be no white space in the argument list between the opening parenthesis and the comma separating the fourth and fifth arguments.

os_relationship_1_1()

ObjectStore allows the user to model binary relationships with pointer-valued (or collection-of-pointer-valued) data members that maintain the referential integrity of their inverse data members. You implement this inverse maintenance by defining an embedded relationship class, which encapsulates the pointer (or collection-of-pointer) so that it can intercept updates to the encapsulated value and perform the necessary inverse maintenance tasks. The encapsulated-pointer values are stored as soft pointers so as to maintain the values across address space release and transactions.

Required include files

To use this macro, you must include the file <os_pse/relat.hh> after including <os_pse/ostore.hh>. If you also include <os_pse/coll.hh>, include <os_pse/relat.hh> after both <os_pse/ostore.hh> and <os_pse/coll.hh>.

The actual value type of a data member with an inverse is a special class whose instances encapsulate the member's apparent value. This implicitly defined class defines `operator =()` (for setting the apparent value), as well as `operator ->()`, `operator *()`, and a conversion operator for converting its instances to instances of the apparent value type (for getting the apparent value). Under most circumstances, these operators make the encapsulating objects transparent.

The implicitly defined class also defines the member functions `getvalue()`, which returns the apparent value, and `setvalue()`, which takes an instance of the apparent value type as argument. These functions can always be used to set and get the member's apparent value explicitly.

This macro is used to declare a single-valued data member with a single-valued inverse data member. The macro call is used instead of the value type in the member declaration.

```
class class-name
{
  ...
  macro-call member-name;
  ...
};
```

Form of the call

`os_relationship_1_1(`*`class`*`,`*`member`*`,`*`inv_class`*`,`*`inv_mem`*`,`*`value_type`*`)`

`class` is the class defining the data member being declared.

`member` is the name of the member being declared.

`inv_class` is the name of the class that defines the inverse member.

`inv_mem` is the name of the inverse member.

`value_type` is the value type of the member being declared.

Caution

The first four macro arguments are used (among other things) to concatenate unique names for the encapsulating relationship class and its accessor functions. The details of macro preprocessing differ from compiler to compiler, and in some cases it is necessary to enter these macro arguments *without white space* to ensure that the argument concatenation will work correctly. There should be no white space in the argument list between the opening parenthesis and the comma separating the fourth and fifth arguments.

os_relationship_1_m()

ObjectStore allows the user to model binary relationships with pointer-valued (or collection-of-pointer-valued) data members that maintain the referential integrity of their inverse data members. You implement this inverse maintenance by defining an embedded relationship class, which encapsulates the pointer (or collection-of-pointer) so that it can intercept updates to the encapsulated value and perform the necessary inverse maintenance tasks. The encapsulated-pointer values are stored as soft pointers so as to maintain the values across address space release and transactions.

Required include files

To use this macro, you must include the file <os_pse/relat.hh> after including <os_pse/ostore.hh> and <os_pse/coll.hh>.

The actual value type of a data member with an inverse is a special class whose instances encapsulate the member's apparent value. This implicitly defined class defines `operator =()` (for setting the apparent value), as well as `operator ->()`, `operator *()`, and a conversion operator for converting its instances to instances of the apparent value type (for getting the apparent value). Under most circumstances, these operators make the encapsulating objects transparent.

The implicitly defined class also defines the member functions `getvalue()`, which returns the apparent value, and `setvalue()`, which takes an instance of the apparent value type as argument. These functions can always be used to set and get the member's apparent value explicitly.

This macro is used to declare a single-valued data member with a many-valued inverse data member. The macro call is used instead of the value type in the member declaration.

```
class class-name
{
  ...
  macro-call member-name;
  ...
};
```

Form of the call

`os_relationship_1_m(class,member,inv_class,inv_mem,value_type)`

`class` is the class defining the data member being declared.

`member` is the name of the member being declared.

`inv_class` is the name of the class that defines the inverse member.

`inv_mem` is the name of the inverse member.

`value_type` is the value type of the member being declared.

Caution

The first four macro arguments are used (among other things) to concatenate unique names for the encapsulating relationship class and its accessor functions. The details of macro preprocessing differ from compiler to compiler, and in some cases it is necessary to enter these macro arguments *without white space* to ensure that the argument concatenation will work correctly. There should be no white space in the argument list between the opening parenthesis and the comma separating the fourth and fifth arguments.

OS_RELATIONSHIP_LINKAGE()

Windows platforms

Specifies the linkage for classes generated by the `os_relationship_`*xxx* macros. This macro can be used with component schemas on Windows platforms. For example, you could define the macro as Microsoft's `__declspec(dllexport)`, which allows one DLL to create a subclass of a class defined in another DLL when there are relationship members.

You must define `OS_RELATIONSHIP_LINKAGE()` before including `<ostore/relat.hh>`. For example:

```
...
#define OS_RELATIONSHIP_LINKAGE __declspec(dllexport)
#include <ostore/relat.hh>
```

If the macro is not defined, the default is blank.

os_relationship_m_1()

ObjectStore allows the user to model binary relationships with pointer-valued (or collection-of-pointer-valued) data members that maintain the referential integrity of their inverse data members. You implement this inverse maintenance by defining an embedded relationship class, which encapsulates the pointer (or collection-of-pointer) so that it can intercept updates to the encapsulated value and perform the necessary inverse maintenance tasks. The encapsulated-pointer values are stored as soft pointers so as to maintain the values across address space release and transactions.

Required include files

To use this macro, you must include the file <os_pse/relat.hh> after including <os_pse/ostore.hh> and <os_pse/coll.hh>.

The actual value type of a data member with an inverse is a special class whose instances encapsulate the member's apparent value. This implicitly defined class defines `operator =()` (for setting the apparent value), as well as `operator ->()`, `operator *()`, and a conversion operator for converting its instances to instances of the apparent value type (for getting the apparent value). Under most circumstances, these operators make the encapsulating objects transparent.

The implicitly defined class also defines the member functions `getvalue()`, which returns the apparent value, and `setvalue()`, which takes an instance of the apparent value type as argument. These functions can always be used to set and get the member's apparent value explicitly.

This macro is used to declare a many-valued data member with a single-valued inverse data member. The macro call is used instead of the value type in the member declaration.

```
class class-name
{
  ...
  macro-call member-name;
  ...
};
```

Form of the call

`os_relationship_m_1(`*`class`*`,`*`member`*`,`*`inv_class`*`,`*`inv_mem`*`,`*`value_type`*`)`

`class` is the class defining the data member being declared.

`member` is the name of the member being declared.

`inv_class` is the name of the class that defines the inverse member.

`inv_mem` is the name of the inverse member.

`value_type` is the value type of the member being declared.

Caution

The first four macro arguments are used (among other things) to concatenate unique names for the encapsulating relationship class and its accessor functions. The details of macro preprocessing differ from compiler to compiler, and in some cases it is necessary to enter these macro arguments *without white space* to ensure that the argument concatenation will work correctly. There should be no white space in the argument list between the opening parenthesis and the comma separating the fourth and fifth arguments.

os_relationship_m_m()

ObjectStore allows the user to model binary relationships with pointer-valued (or collection-of-pointer-valued) data members that maintain the referential integrity of their inverse data members. You implement this inverse maintenance by defining an embedded relationship class, which encapsulates the pointer (or collection-of-pointer) so that it can intercept updates to the encapsulated value and perform the necessary inverse maintenance tasks. The encapsulated-pointer values are stored as soft pointers so as to maintain the values across address space release and transactions.

Required include files

To use this macro, you must include the file <os_pse/relat.hh> after including <os_pse/ostore.hh> and <os_pse/coll.hh>.

The actual value type of a data member with an inverse is a special class whose instances encapsulate the member's apparent value. This implicitly defined class defines `operator =()` (for setting the apparent value), as well as `operator ->()`, `operator *()`, and a conversion operator for converting its instances to instances of the apparent value type (for getting the apparent value). Under most circumstances, these operators make the encapsulating objects transparent.

The implicitly defined class also defines the member functions `getvalue()`, which returns the apparent value, and `setvalue()`, which takes an instance of the apparent value type as argument. These functions can always be used to set and get the member's apparent value explicitly.

This macro is used to declare a many-valued data member with a many-valued inverse data member. The macro call is used instead of the value type in the member declaration.

```
class class-name
{
  ...
  macro-call member-name;
  ...
};
```

Form of the call

`os_relationship_m_m(`*`class`*`,`*`member`*`,`*`inv_class`*`,`*`inv_mem`*`,`*`value_type`*`)`

`class` is the class defining the data member being declared.

`member` is the name of the member being declared.

`inv_class` is the name of the class that defines the inverse member.

`inv_mem` is the name of the inverse member.

`value_type` is the value type of the member being declared.

Caution

The first four macro arguments are used (among other things) to concatenate unique names for the encapsulating relationship class and its accessor functions. The details of macro preprocessing differ from compiler to compiler, and in some cases it is necessary to enter these macro arguments *without white space* to ensure that the argument concatenation will work correctly. There should be no white space in the argument list between the opening parenthesis and the comma separating the fourth and fifth arguments.

Index

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V