OBJECTSTORE

COLLECTIONS C++ API REFERENCE

RELEASE 5.1

March 1998

ObjectStore Collections C++ API Reference

ObjectStore Release 5.1, March 1998

Object Design, Inc.
Twenty Five Mall Road
Burlington, MA 01803-4194

Part number: SW-OS-DOC-CRF-510

Contents

Preface

Purpose

The *ObjectStore Collections C++ API Reference* provides a reference on C++ programming interfaces to ObjectStore for collections, queries, and indexes. This book supports ObjectStore Release 5.1.

Audience

This book assumes the reader is experienced with C++.

Scope

Information in this book assumes that ObjectStore is installed and configured.

How This Book Is Organized

The manual has five chapters and an appendix:

It begins with an introductory chapter. Each chapter after that covers a type of interface, including classes, representation types, macros, user-supplied functions, and the C library interface.

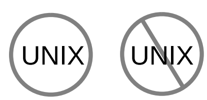
Within each chapter, material is organized alphabetically.

Topic	***Location***
Database services	Chapter 1, Introduction, on page 1
The C++ API for ObjectStore collections, queries, and indexes	Chapter 2, Collection, Query, and Index Classes, on page 3
Representation types	Chapter 3, Representation Types, on page 245
System-supplied macros User-defined functions	Chapter 4, Macros and User-Defined Functions, on page 265
C library interface	Chapter 5, C Library Interface, on page 309

Topic	*Location*
Exceptions	Appendix, Predefined TIX Exceptions, on page 335

Notation Conventions

This document uses the following conventions:

Convention	***Meaning***
Bold	Bold typeface indicates user input or code.
Sans serif	Sans serif typeface indicates system output.
Italic sans serif	Italic sans serif typeface indicates a variable for which you must supply a value. This most often appears in a syntax line or table.
Italic serif	In text, italic serif typeface indicates the first use of an important term.
[ ]	Brackets enclose optional arguments.
{ *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 indicate that you can repeat the immediately previous item. In examples, they also indicate omissions.
	Indicates that the operating system named inside the circle supports or does not support the feature being discussed.

ObjectStore C++ Release 5.1 Documentation

The ObjectStore Release 5.1 documentation is chiefly distributed on line in web-browsable format. If you want to order printed books, contact your Object Design sales representative.

Your use of ObjectStore documentation depends on your role and level of experience with ObjectStore. You can find an overview description of each book in the ObjectStore documentation set at URL **http://www.objectdesign.com**. Select **Products** and then select **Product Documentation** to view these descriptions.

Internet Sources for More Information

World Wide Web

Object Design's support organization provides a number of information resources. These are available to you through a web browser such as Mosaic or Netscape. You can obtain information by accessing the Object Design home page with the URL **http://www.objectdesign.com**. Select **Technical Support**. Select **Support Communications** for detailed instructions about different methods of obtaining information from support.

Internet gateway

You can obtain such information as frequently asked questions (FAQs) from Object Design's Internet gateway machine as well as from the web. This machine is called **ftp.objectdesign.com** and its Internet address is 198.3.16.26. You can use **ftp** to retrieve the FAQs from there. Use the login name **odiftp** and the password obtained from **patch-info**. This password also changes monthly, but you can automatically receive the updated password by subscribing to **patch-info**. See the **README** file for guidelines for using this connection. The FAQs are in the subdirectory **./FAQ**. This directory contains a group of subdirectories organized by topic. The file **./FAQ/FAQ.tar.Z** is a compressed **tar** version of this hierarchy that you can download.

Automatic email notification

In addition to the previous methods of obtaining Object Design's latest patch updates (available on the **ftp** server as well as the Object Design Support home page) you can now automatically be notified of updates. To subscribe, send email to **patch-info-request@objectdesign.com** with the keyword **SUBSCRIBE patch-info <***your siteid***>** in the body of your email. This will subscribe you to Object Design's patch information server daemon that automatically provides site access information and notification of other changes to the on-line support services. Your site ID is listed on any shipment from Object Design, or you can contact your Object Design Sales Administrator for the site ID information.

Training

If you are in North America, for information about Object Design's educational offerings, or to order additional documents, call 781.674.5000, Monday through Friday from 8:30 AM to 5:30 PM Eastern Time.

If you are outside North America, call your Object Design sales representative.

Your Comments

Object Design welcomes your comments about ObjectStore documentation. Send your feedback to **support@objectdesign.com**. To expedite your message, begin the subject with **Doc:**. For example:

Subject: Doc: Incorrect message in description of objectstore::foo() in the reference manual

You can also fax your comments to 781.674.5440.

Chapter 1
Introduction

This document describes the C and C++ application programming interface to the functionality provided by the collections, queries, and indexes database services for the ObjectStore object-oriented database management system.

For task-oriented information, see the *ObjectStore C++ API User Guide* and the *ObjectStore Advanced C++ API User Guide*.

For reference information on other database services and interfaces see the *ObjectStore C++ API Reference*.

Query Processing

The query processing database service provides support for associative data retrieval, such as lookup by name or ID number.

Data Access

Many application types require two forms of data access: navigational access and associative access. Navigation accesses data by following pointers contained in data structure fields. In C++, the data member access syntax supports navigational data access. Associative access, on the other hand, is the lookup of those data structures whose field values satisfy a certain condition (for example, lookup of an object by name or ID number). ObjectStore supports associative access, or query, through member functions in the ObjectStore class library.

Collections

Queries involve *collections,* which are objects such as sets, bags, or lists that serve to group together other objects. ObjectStore provides a library of collection classes. These classes provide the data structures for representing such collections, encapsulated by member functions that support various forms of collection manipulation, such as element insertion and removal. Retrieval of a given collection's elements for examination or processing one at a time is supported through the use of a cursor class.

Query Optimizer

Queries return a collection containing those elements of a given collection that satisfy a specified condition. They can be executed with an optimized search strategy, formulated by the ObjectStore *query optimizer*. The query optimizer maintains indexes into collections based on user-specified keys, that is, data members, or data members of data members, and so on.

Indexes

By using these indexes, implemented as B-trees or hash tables, the programmer can minimize the number of objects examined in response to a query. Formulation of optimization strategies is performed automatically by the system. Index maintenance can also be automatic — the programmer need only specify the desired index keys.

Chapter 2
Collection, Query, and Index Classes

Classes

This chapter presents the following classes related to collections, queries, and indexes:

os_Array

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

An array, like a list (see the class **os_List** on page 186), is an ordered collection. Arrays always provide access to collection elements in constant time. That is, for all allowable representations of an **os_Array**, 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. In addition, the array **create()** functions have additional arguments that allow specification of the initial array cardinality and fill value (the initial value to put in each slot).

By default, 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 18). This means that when specifying **os_Array** as a function's formal parameter, or as the type of a variable or data member, you must specify the parameter (the array's *element type*). This is accomplished by appending to **os_Array** the name of the element type enclosed in angle brackets, < >:

os_Array<*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 collection type, such as an array, must be a pointer type (for example, **employee***).

Create arrays with the member **create()** or, for stack-based or embedded collections, with a constructor. Do not use **new** to create arrays.

Required header files

Programs using arrays must include the header file **<ostore/coll.hh>** after including **<ostore/ostore.hh>**.

Required libraries

Programs using arrays must link with the library file **oscol.lib** (UNIX platforms) or **oscol.ldb** (Windows platforms).

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.

Below are two tables. The first table 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*
add_index	**(const os_index_path&, os_int32 options, os_database*)**	**void**	**os_collection**
	(const os_index_path&, os_int32 options, os_segment* = 0)	**void**	
	(const os_index_path&, os_segment* = 0)	**void**	
	(const os_index_path&, os_database*)	**void**	
cardinality	**( ) const**	**os_unsigned_int32**	**os_collection**
change_behavior	**(os_unsigned_int32 behavior_enums, os_int32 = verify_enum)**	**void**	**os_collection**

Name	*Arguments*	*Returns*	*Defined By*
change_rep	**(os_unsigned_int32 expected_size const os_coll_rep_descriptor *policy = 0, os_int32 retain_enum = dont_associate_policy)**	**void**	**os_collection**
clear	**( )**	**void**	**os_collection**
contains	**(const E) const**	**os_int32**	**os_Collection**
count	**(const E) const**	**os_int32**	**os_Collection**
create (static)	**(os_database *db, os_unsigned_int32 behavior_enums = 0, os_int32 expected_size = 0, const os_coll_rep_descriptor* = 0, os_int32 retain_enum = dont_associate_policy, os_unsigned_int32 cardinality = 0, E fill_value = 0)**	**os_Array<E>&**	**os_Array**
	(os_segment* seg, os_unsigned_int32 behavior_enums = 0, os_int32 expected_size = 0, const os_coll_rep_descriptor* = 0, os_int32 retain_enum = dont_associate_policy, os_unsigned_int32 cardinality = 0, E fill_value = 0)	**os_Array<E>&**	
	(os_object_cluster *clust, os_unsigned_int32 behavior_enums = 0, os_int32 expected_size = 0, const os_coll_rep_descriptor* = 0, os_int32 retain_enum = dont_associate_policy, os_unsigned_int32 cardinality = 0, E fill_value = 0)	**os_Array<E>&**	
	(void *proximity, os_unsigned_int32 behavior_enums = 0, os_int32 expected_size = 0, const os_coll_rep_descriptor* = 0, os_int32 retain_enum = dont_associate_policy, os_unsigned_int32 cardinality = 0, E fill_value = 0)	**os_Array<E>&**	
default_behavior (static)	**( )**	**os_unsigned_int32**	**os_Array**
destroy (static)	**(os_Array<E>&)**	**void**	**os_Array**
drop_index	**(const os_index_path&)**	**void**	**os_collection**

Name	Arguments	Returns	Defined By
empty	( )	os_int32	os_collection
exists	(char *element_type_name, char *query_string, os_database *schema_database = 0, char *file os_unsigned_int32 line)	os_int32	os_collection
	(const os_bound_query&) const	os_int32	
get_behavior	( ) const	os_unsigned_int32	os_collection
get_rep	( ) const	os_coll_rep_ descriptor&	os_collection
has_index	(const os_index_path&, os_int32 index_option_enums) const	os_int32	os_collection
initialize (static)	( )	void	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
multi_trans_add_ index	static void multi_trans_add_index(os_reference c, const os_index_path & p, os_int32 index_options, os_segment * index_seg, os_segment * scratch_seg, os_unsigned_int32 num_per_trans)		os_collection
multi_trans_drop_ index	static void multi_trans_drop_index(os_reference c, const os_index_path & p, os_segment * scratch_seg, os_unsigned_int32 num_per_trans)		os_collection
only	( ) const	E	os_Collection
operator os_Bag<E>&	( )		os_Collection

Name	*Arguments*	*Returns*	*Defined By*
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 ==	**(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**	
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**	
operator =	**(const os_Array<E>&) const**	**os_Array<E>&**	**os_Array**
	(const os_Collection<E>&) const	**os_array&**	
	(E) const	**os_array**	
operator |=	**(const os_Collection<E>&) const**	**os_Array<E>&**	**os_Array**
	(E) const	**os_Array<E>&**	

<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</td><td>os_Collection<E>&</td><td>os_Collection</td></tr>
<tr><td></td><td>(E) const</td><td>os_Collection<E>&</td><td></td></tr>
<tr><td>operator &=</td><td>(const os_Collection<E>&) const</td><td>os_Array<E>&</td><td>os_Array</td></tr>
<tr><td></td><td>(E) const</td><td>os_Array<E>&</td><td></td></tr>
<tr><td>operator &</td><td>(const os_Collection<E>&) const</td><td>os_Collection<E>&</td><td>os_Collection</td></tr>
<tr><td></td><td>(E) const</td><td>os_Collection<E>&</td><td></td></tr>
<tr><td>operator –=</td><td>(const os_Collection<E>&) const</td><td>os_Array<E>&</td><td>os_Array</td></tr>
<tr><td></td><td>(E) const</td><td>os_Array<E>&</td><td></td></tr>
<tr><td>operator -</td><td>(const os_Collection<E>&) const</td><td>os_Collection<E>&</td><td>os_Collection</td></tr>
<tr><td></td><td>(E) const</td><td>os_Collection<E>&</td><td></td></tr>
<tr><td>os_Array<E></td><td>( )</td><td></td><td>os_Array</td></tr>
<tr><td></td><td>(os_collection_size expected_size)</td><td></td><td></td></tr>
<tr><td></td><td>(const os_Array<E>&)</td><td></td><td></td></tr>
<tr><td></td><td>(const const os_Collection<E>&)</td><td></td><td></td></tr>
<tr><td></td><td>(os_unsigned_int32 card,
E fill_value)</td><td></td><td></td></tr>
<tr><td>pick</td><td>( ) const</td><td>E</td><td>os_Collection</td></tr>
<tr><td></td><td>(const os_index_path&,
const os_coll_range&) const</td><td>E</td><td></td></tr>
<tr><td>query</td><td>(char *element_type_name,
char *query_string,
os_database *schema_database = 0,
char *file,
os_unsigned_int32 line,
os_boolean dups) const</td><td>os_Collection<E>&</td><td>os_Collection</td></tr>
<tr><td></td><td>(const os_bound_query&,
os_boolean dups) const</td><td>os_Collection<E>&</td><td></td></tr>
<tr><td>query_pick</td><td>(char *element_type_name,
char *query_string,
os_database *schema_database = 0,
char *filename,
os_unsigned_int32 line) const</td><td>E</td><td>os_Collection</td></tr>
<tr><td></td><td>(const os_bound_query&) const</td><td>E</td><td></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></td><td>(os_unsigned_int32)</td><td>void</td><td></td></tr>
</table>

Name	*Arguments*	*Returns*	*Defined By*
remove_first	**(const E&)**	**os_int32**	**os_Collection**
	( )	**E**	
remove_last	**(const E&)**	**os_int32**	**os_Collection**
	( )	**E**	
replace_at	**(const E, const os_Cursor<E>&)**	**E**	**os_Collection**
	(const E, os_unsigned_int32)	**E**	
retrieve	**(os_unsigned_int32) const**	**E**	**os_Collection**
	(const os_Cursor<E>&) const	**E**	
retrieve_first	**( ) const**	**E**	**os_Collection**
	(const E&) const	**os_int32**	
retrieve_last	**( ) const**	**E**	**os_Collection**
	(const E&) const	**os_int32**	
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**
associate_policy	**os_collection**
dont_associate_policy	**os_collection**
dont_verify	**os_collection**
EQ	**os_collection**
GT	**os_collection**
LT	**os_collection**
maintain_cursors	**os_collection**
maintain_order	**os_collection**
pick_from_empty_returns_null	**os_collection**
signal_cardinality	**os_collection**
signal_duplicates	**os_collection**

Name	Inherited From
unordered	os_collection
verify	os_collection

os_Array::create()

```
static os_Array<E> &create(
    os_database *db,
    os_unsigned_int32 behavior_enums = 0,
    os_int32 expected_size = 0,
    const os_coll_rep_descriptor *rep_policy = 0,
    os_int32 retain_enum = dont_associate_policy,
    os_unsigned_int32 cardinality = 0,
    E  fill_value = 0
);
```

Creates a new array in the database pointed to by **db**. If the transient database is specified, the array is allocated in transient memory.

Properties

A new array has the following default properties:

Property	*Enumerator That Controls Behavior*
Its entries are ordered.	**os_collection::maintain_order** (required)
Duplicate elements are allowed.	**os_collection::allow_duplicates** (on by default)
Null pointers can be inserted.	**os_collection::allow_nulls** (required)
It has array semantics.	**os_collection::be_an_array** (required)

An array can also have these behaviors:

Behavior	*Enumerators*
With **pick()** returns null from an empty array. When this behavior is not specified, err_coll_empty is raised.	**os_collection::pick_from_empty_ returns_null**

Behavior	***Enumerators***
Signals when an attempt is made to insert a duplicate element into array for **allow_duplicates** is not in effect.	**os_collection::signal_duplicates**
Maintains the position of a cursor while elements are being inserted or removed from an array.	**os_collection::maintains_cursors**

By default a new array also has the following properties:

- Performing **pick()** on an empty array raises err_coll_empty.
- 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.

Using the **behavior_enums** argument, you can customize the properties of new arrays with regard to these two properties.

See Customizing Collection Behavior in *ObjectStore Advanced C++ API User Guide* for further information.

Representation policy

The default representation policy for arrays is as follows:

- An array created as an embedded object has the representation of **os_tiny_array** (0 to 4 elements). An embedded array becomes out of line and mutates to an **os_chained_list** when the fifth element is inserted.
- An array created with **::create** with cardinality <= 20 is represented as an **os_chained_list**.
- Once the array grows past 20, its representation is **os_packed_list.** (see the description in Chapter 4, Advanced Collections, of *ObjectStore Advanced C++ API User Guide*).

If you want a new array presized for a different cardinality, supply the **expected_size** argument explicitly.

Note that **expected_size** determines the initial representation. So, for example, if **expected_size** is 21, **os_packed_list** is used for the array's entire lifetime (unless you use **change_rep()**).

If you want to customize the representation of a new array, pass an **os_rep** as the **rep_policy** argument and pass the enumerator

os_collection::dont_associate_policy as the **retain** argument, or else pass an **os_rep_policy** as the **policy** argument and pass the enumerator **os_collection::associate_policy** as the **retain** argument. See the class **os_rep** on page 224. See also Customizing Collection Behavior in *ObjectStore Advanced C++ API User Guide*.

cardinality and **fill_value**

cardinality and **fill_value** specify the number of slots in the new array, and the value to occupy all slots initially. The value specified by the **cardinality** argument must be less than or equal to the **expected_size**

can also affect the underlying collection representation; the larger of the two values (the values for **cardinality** and **expected_size**) takes precedence.

```
static os_Array<E> &create(
    os_segment * seg,
    os_unsigned_int32 behavior = 0,
    os_int32 expected_size = 0,
    const os_coll_rep_descriptor *rep_policy = 0,
    os_int32 retain = dont_associate_policy,
    os_unsigned_int32 cardinality = 0,
    E fill_value = 0
);
```

Creates a new array in the segment pointed to by **seg**. If the transient segment is specified, the array is allocated in transient memory. The rest of the arguments are just as described above.

```
static os_Array<E> &create(
    os_object_cluster *clust,
    os_unsigned_int32 behavior = 0,
    os_int32 expected_size = 0,
    const os_coll_rep_descriptor *rep_policy = 0,
    os_int32 retain = dont_associate_policy,
    os_int32 cardinality = 0,
    E fill_value = 0
);
```

Creates a new array in the object cluster pointed to by **clust**. The rest of the arguments are just as described above.

```
static os_Array<E> &create(
    void * proximity,
    os_unsigned_int32 behavior = 0,
    os_int32 expected_size = 0,
    const os_coll_rep_descriptor *rep_policy = 0,
    os_int32 retain = dont_associate_policy,
    os_int32 cardinality = 0,
    E fill_value = 0
```

);

Creates a new array in the segment occupied by the object pointed to by **proximity**. If the object is part of an object cluster, the new array is allocated in that cluster. If the specified object is transient, the array is allocated in transient memory. The rest of the arguments are just as described above.

os_Array::default_behavior()

static os_unsigned_int32 default_behavior();

Returns a bit pattern indicating this type's default behavior, which is **maintain_order**, **allow_duplicates**, **allow_nulls,** and **be_an_array.**

os_Array::destroy()

static void destroy(os_Array<E>&);

Deletes the specified array and deallocates associated storage. This is the same as calling delete on a pointer to an **os_Array**.

Assignment Operator Semantics

Note: The assignment operator semantics are described for the following functions in terms of insert operations into the target collection. Describing the semantics in terms of insert operations serves to illustrate how duplicate, null, and order semantics are enforced. The actual implementation of the assignment might be quite different, while still maintaining the associated semantics.

os_Array::operator =()

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

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

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

Copies the contents of the array **s** into the target array and returns the target array. The copy is performed by effectively clearing the target, iterating over the source array, and inserting each element into the target array. The iteration is ordered if the source array 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.

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

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

os_Array::os_Array()

os_Array();

Returns an empty array.

os_Array(os_collection_size);

The user should pass an **os_int32** as the **os_collection_size** argument. Returns an empty array whose initial implementation is based on the expectation that the specified **os_int32** indicates the approximate usual cardinality of the array, once 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 card, E fill_value);

Returns an array with cardinality **card**, all of whose elements are **fill_value**.

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 198), is an ordered collection. Unlike other ordered collections, however, arrays 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. In addition, the array **create()** functions have additional arguments that allow specification of the initial array cardinality and fill value (the initial value to put in each slot). By default, 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.

The class **os_array** is nonparameterized. For the parameterized version of this class, see **os_Array** on page 5.

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

Create arrays with the member **create()** or, for stack-based or embedded arrays, with a constructor. Do not use **new** to create arrays.

Required header files

Any program using arrays must include the header file **<ostore/coll.hh>** after including **<ostore/ostore.hh>**.

Required libraries

Programs using arrays must link with the library file **oscol.lib** (UNIX platforms) or **oscol.ldb** (Windows platforms).

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.

Below are two tables. The first table 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*
add_index	**(const os_index_path&, os_int32 = options, os_database* = 0)**	**void**	**os_collection**
	(const os_index_path&, os_int32 = options, os_segment* = 0)	**void**	
	(const os_index_path&, os_database* = 0)	**void**	
	(const os_index_path&, os_segment* = 0)	**void**	
cardinality	**( ) const**	**os_unsigned_int32**	**os_collection**
change_behavior	**(os_unsigned_int32 behavior enums, os_int32 = verify_enum)**	**void**	**os_collection**
change_rep	**(os_unsigned_int32 expected_size, const os_coll_rep_descriptor *policy = 0, os_int32 retain_enum = dont_associate_policy)**	**void**	**os_collection**
clear	**( )**	**void**	**os_collection**
contains	**(const void*) const**	**os_int32**	**os_collection**
count	**(const void*) const**	**os_int32**	**os_collection**
create (static)	**(os_database *db, os_unsigned_int32 behavior_enums = 0, os_int32 expected_size = 0, const os_coll_rep_descriptor* = 0, os_int32 retain_enum = dont_associate_policy, os_unsigned_int32 cardinality = 0, void* fill_value = 0)**	**os_array&**	**os_array**
	(os_segment *seg, os_unsigned_int32 behavior_enums = 0, os_int32 expected_size = 0, const os_coll_rep_descriptor* = 0, os_int32 retain_enum = dont_associate_policy, os_unsigned_int32 cardinality = 0, void* fill_value = 0)	**os_array&**	

Name	Arguments	Returns	Defined By
	(os_object_cluster *clust, os_unsigned_int32 behavior_enums = 0, os_int32 expected_size = 0, const os_coll_rep_descriptor* = 0, os_int32 retain_enum = dont_associate_policy, os_unsigned_int32 cardinality = 0, void* fill_value = 0)	os_array&	
	(void *proximity, os_unsigned_int32 behavior_enums = 0, os_int32 expected_size = 0, const os_coll_rep_descriptor* = 0, os_int32 retain_enum = dont_associate_policy, os_unsigned_int32 cardinality = 0, void* fill_value = 0)	os_array&	
default_behavior (static)	( )	os_unsigned_int32	os_array
destroy (static)	(os_array&)	void	os_array
drop_index	(const os_index_path&)	void	os_collection
empty	( ) const	os_int32	os_collection
exists	(char *element_type_name, char *query_string, os_database *schema_database = 0, char *file os_unsigned_int32 line)	os_int32	os_collection
	(const os_bound_query&) const	os_int32	
get_behavior	( ) const	os_unsigned_int32	os_collection
get_rep	( ) const	os_coll_rep_ descriptor&	os_collection
has_index	(const os_index_path&, os_int32 index_option_enums) const	os_int32	os_collection
insert	(const void*)	void	os_collection
insert_after	(const void*, const os_cursor&)	void	os_collection
	(const void*, os_unsigned_int32)	void	
insert_before	(const void*, const os_cursor&)	void	os_collection
	(const void*, os_unsigned_int32)	void	

Name	*Arguments*	*Returns*	*Defined By*
insert_first	**(const void*)**	**void**	**os_collection**
insert_last	**(const void*)**	**void**	**os_collection**
multi_trans_add_ index	**static void multi_trans_add_index(os_reference c, const os_index_path & p, os_int32 index_options, os_segment * index_seg, os_segment * scratch_seg, os_unsigned_int32 num_per_trans)**		**os_collection**
multi_trans_ drop_index	**static void multi_trans_drop_index(os_reference c, const os_index_path & p, os_segment * scratch_seg, os_unsigned_int32 num_per_trans)**		**os_collection**
only	**( ) const**	**void***	**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&	**( )**		**os_collection**
operator const os_set&	**( ) const**		**os_collection**
operator ==	**(const os_collection&) const**	**os_int32**	**os_collection**
	(const void*) const	**os_int32**	
operator !=	**(const os_collection&) const**	**os_int32**	**os_collection**
	(const void*) const	**os_int32**	
operator <	**(const os_collection&) const**	**os_int32**	**os_collection**
	(const void*) const	**os_int32**	
operator <=	**(const os_collection&) const**	**os_int32**	**os_collection**
	(const void*) const	**os_int32**	
operator >	**(const os_collection&) const**	**os_int32**	**os_collection**
	(const void*) const	**os_int32**	
operator >=	**(const os_collection&) const**	**os_int32**	**os_collection**
	(const void*) const	**os_int32**	

Name	***Arguments***	***Returns***	***Defined By***
operator =	**(const os_array&) const**	**os_array&**	**os_array**
	(const os_collection&) const	**os_array&**	
	(const void*) const	**os_array**	
operator |=	**(const os_collection&) const**	**os_array&**	**os_array**
	(const void*) const	**os_array&**	
operator |	**(const os_collection&) const**	**os_array&**	**os_array**
	(const void*) const	**os_array&**	
operator &=	**(const os_collection&) const**	**os_array&**	**os_array**
	(const void*) const	**os_array&**	
operator &	**(const os_collection&) const**	**os_array&**	**os_array**
	(const void*) const	**os_array&**	
operator –=	**(const os_collection&) const**	**os_array&**	**os_array**
	(const void*) const	**os_array&**	
operator -	**(const os_collection&) const**	**os_array&**	**os_array**
	(const void*) const	**os_array&**	
os_array	**( )**		**os_array**
	(os_int32 expected_size)		
	(const os_array&)		
	(const os_collection&)		
	(os_unsigned_int32 card, void *fill_value)		
pick	**( ) const**	**void***	**os_collection**
query	**(char *element_type_name, char *query_string, os_database *schema_database = 0, char *file, os_unsigned_int32 line, os_boolean dups) const**	**os_collection&**	**os_collection**
	(const os_bound_query&, os_boolean dups) const	**os_collection&**	

Name	*Arguments*	*Returns*	*Defined By*
query_pick	**(char *element_type_name, char *query_string, os_database *schema_database = 0, char *filename, os_unsigned_int32 line) const**	**void***	**os_collection**
	(const os_bound_query&) const	**void***	
remove	**(const void*)**	**os_int32**	**os_collection**
remove_at	**(const os_cursor&)**	**void**	**os_collection**
	(os_unsigned_int32)	**void**	
remove_first	**(const void*&)**	**os_int32**	**os_collection**
	( )	**void***	
remove_last	**(const void*&)**	**os_int32**	**os_collection**
	( )	**void***	
replace_at	**(const void*, const os_cursor&)**	**void***	**os_collection**
	(const void*, os_unsigned_int32)	**void***	
retrieve	**(os_unsigned_int32) const**	**void***	**os_collection**
	(const os_cursor&) const	**void***	
retrieve_first	**( ) const**	**void***	**os_collection**
	(const void*&) const	**os_int32**	
retrieve_last	**( ) const**	**void***	**os_collection**
	(const void*&) const	**os_int32**	
set_cardinality	**(os_unsigned_int32 new_card, void *fill_value)**	**void**	**os_array**

os_array enumerators

The following table lists the enumerators inherited by **os_array** from **os_collection**.

Name	*Inherited From*
allow_duplicates	**os_collection**
allow_nulls	**os_collection**
associate_policy	**os_collection**
dont_associate_policy	**os_collection**
dont_verify	**os_collection**
EQ	**os_collection**

Name	*Inherited From*
GT	**os_collection**
LT	**os_collection**
maintain_cursors	**os_collection**
maintain_order	**os_collection**
pick_from_empty_returns_null	**os_collection**
signal_cardinality	**os_collection**
signal_duplicates	**os_collection**
unordered	**os_collection**
verify	**os_collection**

os_array::create()

```
static os_array &create(
    os_database *db,
    os_unsigned_int32 behavior = 0,
    os_int32 expected_size = 0,
    const os_coll_rep_descriptor *rep_policy = 0,
    os_int32 retain = dont_associate_policy,
    os_unsigned_int32 cardinality = 0,
    void *fill_value = 0
);
```

Creates a new array in the database pointed to by **db**. If the transient database is specified, the array is allocated in transient memory.

Properties

Every array has the following properties:

- Its entries are ordered.
- Duplicate elements are allowed.
- Null pointers can be inserted.

By default a new array also has the following properties:

- Performing **pick()** on an empty result of querying the array raises err_coll_empty.
- 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.

See also Customizing Collection Behavior in *ObjectStore Advanced C++ API User Guide*.

By default, arrays are presized with a representation suitable for cardinality 20 or less. If you want a new collection presized for a different cardinality, supply the **expected_size** argument explicitly.

If you want to customize the representation of a new collection, see Customizing Collection Representation in *ObjectStore Advanced C++ API User Guide*.

Representation policy

The default representation policy for arrays is as follows:

- As the array grows from 0 to 15, the representation is **os_chained_list** (see a description in Chapter 4, Advanced Collections, of *ObjectStore Advanced C++ API User Guide*).
- Once the array grows past 15, **os_packed_list** is used.

Note that **expected_size** determines the initial representation. So, for example, if **expected_size** is 21, **os_packed_list** is used for the array's entire lifetime (unless you use **change_rep()**).

cardinality and **fill_value**

cardinality and **fill_value** specify the number of slots in the new array, and the value to occupy all slots initially.

```
static os_array &create(
    os_segment * seg,
    os_unsigned_int32 behavior = 0,
    os_int32 expected_size = 0,
    const os_coll_rep_descriptor *rep_policy = 0,
    os_int32 retain = dont_associate_policy,
    os_unsigned_int32 cardinality = 0,
    void *fill_value = 0
);
```

Creates a new array in the segment pointed to by **seg**. If the transient segment is specified, the array is allocated in transient memory. The rest of the arguments are just as described above.

```
static os_array &create(
    os_object_cluster *clust,
    os_unsigned_int32 behavior = 0,
    os_int32 expected_size = 0,
    const os_coll_rep_descriptor *rep_policy = 0,
    os_int32 retain = dont_associate_policy,
    os_unsigned_int32 cardinality = 0,
    void *fill_value = 0
);
```

Creates a new array in the object cluster pointed to by **clust**. The rest of the arguments are just as described above.

```
static os_array &create(
    void * proximity,
    os_unsigned_int32 behavior = 0,
    os_int32 expected_size = 0,
    const os_coll_rep_descriptor *rep_policy = 0,
    os_int32 retain = dont_associate_policy,
    os_unsigned_int32 cardinality = 0,
    void *fill_value = 0
);
```

Creates a new array in the segment occupied by the object pointed to by **proximity**. If the object is part of an object cluster, the new array is allocated in that cluster. If the specified object is transient, the array is allocated in transient memory. The rest of the arguments are just as described above.

os_array::default_behavior()

static os_unsigned_int32 default_behavior();

Returns a bit pattern indicating this type's default behavior.

os_array::destroy()

static void destroy(os_array&);

Deletes the specified array and deallocates associated storage.

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 collection 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.

Note on assignment operator semantics

The assignment operator semantics are described throughout the *ObjectStore Collections C++ API Reference* in terms of insert operations into the target array. Describing the semantics in terms of insert operations serves to illustrate how duplicate, null, and order semantics are enforced. The actual implementation of the assignment might be quite different, while still maintaining the associated semantics.

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.

Note: Assignment operator semantics are described in terms of insert operations into the target array. Note, however that the actual implementation might be different, while still maintaining the associated semantics.

os_array::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 nulls, the result does. The result allows duplicates and does not maintain cursors or signal duplicates.

os_collection &operator |(const void *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 nulls, the result does. The result allows duplicates and does not maintain cursors or signal duplicates.

Note: Assignment operator semantics are described in terms of insert operations into the target array. Note, however that the actual implementation might be different, while still maintaining the associated semantics.

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. It 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. It returns the target collection.

Note: Assignment operator semantics are described in terms of insert operations into the target array. Note, however that the actual implementation might be different, while still maintaining the associated semantics.

os_array::operator &()

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

Copies the contents of **this** into a new array, **a**, and then performs **a &= s**. The new array, **a**, is then returned. If either operand allows nulls, the result does. The result allows duplicates and does not maintain cursors or signal duplicates.

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

Creates a new array and copies the element **e** into it if **this-> e** returns true. Behavior bits match those of the original array. That is, if **this** allows nulls, the result does. The result allows duplicates and does not maintain cursors or signal duplicates.

Note: Assignment operator semantics are described in terms of insert operations into the target array. Note, however that the actual implementation might be different, while still maintaining the associated semantics.

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. It 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. It returns the target collection.

Note: Assignment operator semantics are described in terms of insert operations into the target array. Note, however that the actual implementation might be different, while still maintaining the associated semantics.

os_array::operator –()

os_array &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 nulls, the result does. The result allows duplicates and does not maintain cursors or signal duplicates.

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

Copies the contents of **this** into a new array and then removes **e** from the array and returns the new array. If **this** allows nulls, the result does. The result allows duplicates and does not maintain cursors or signal duplicates.

Note: Assignment operator semantics are described in terms of insert operations into the target array. Note, however that the actual implementation might be different, while still maintaining the associated semantics.

os_array::os_array()

os_array();

Returns an empty array.

os_array(os_collection_size);

The user should pass an **os_int32** as the **os_collection_size** argument. Returns an empty array whose initial implementation is based on the expectation that the specified **os_int32** indicates the approximate usual cardinality of the array, once 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 card, void *fill_value);

Returns an array with cardinality **card**, all of whose elements are **fill_value**.

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_backptr

If there is a possibility that a data member of an object could be modified and this data member is used in an index, then index maintenance must occur before and after the update. This is possible only if the object has an **os_backptr** data member and appropriate index maintenance occurs. The **os_backptr** is an object that allows the object to point back to all indexes it participates in.

The **os_backptr** declaration must appear *before* the declaration of the data members intended to be indexable. Note that you must define at most one data member of type **os_backptr**. With one such member, all members (data and function) of the class declared after it are established as indexable.

When an element is inserted or removed from a collection that has an index on it, implicit index maintenance occurs, whether the element has an **os_backptr** or not. Another possibility for update of data members that participate in an index is that you remove the element from the collection, update the data member, then insert it back into the collection. This then performs the appropriate index maintenance. Using this scenario avoids the requirement of having an **os_backptr** in your object. Because an **os_backptr** data member takes up twelve bytes and points back to the indexes, using an **os_backptr** data member might not be desirable in some cases, such as when using reference-based indexes.

ObjectStore supports inheritance of the **os_backptr** data member provided that the member is inherited from a base class along the leftmost side of the type inheritance lattice and provided that the leftmost base class is not a virtual base class (directly or through inheritance). In all other cases, the user must define a data member of type **os_backptr** directly in the class defining the members desired to be indexable.

The **os_backptr** member is used internally by ObjectStore for index maintenance associated with indexable members defined using the **os_indexable_member()** macro or with data members for which you are doing manual index maintenance (**break_link()** or **make_link()**). An example of where manual index maintenance is required is if the data member is a pointer and what gets updated is what is pointed to rather than the pointer value. The member functions described below can also be used explicitly by the user for index maintenance associated with indexable members defined without the **os_indexable_member()** macro.

The **break_link()** function should be invoked to break the index association before a modification to an indexable data member. This removes an entry from the index. In addition, after an indexable member has been given a new value, **make_link()** should be invoked to bring the index up to date. This inserts a new entry into the index, indexing the object by its new member value. You can ensure that this happens by encapsulating these calls in a member function for setting the value of the indexable member. The function should call **break_link()**, assign the new value to the member, and then call **make_link()**. There are also special considerations for index maintenance when member functions are incurred.

member functions that will be used in query strings or index paths

For examples, see User-Controlled Index Maintenance with an os_backptr in the *ObjectStore Advanced C++ API User Guide*.

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.

os_backptr::break_link()

void break_link(void *, void*, os_int32 = 0) const;

Removes an entry from the index associated with the indexable data member pointed to by the **void*** arguments. For a class, **C,** with **os_backptr** member **C::b** and indexable member **m**, the **void*** arguments should both be

&m

The **os_int32** argument should be

os_index(C,b) – os_index(C,m)

The **this** pointer points to the **os_backptr** for the object indexed by the removed entry. The removed entry indexes the object by the value of the specified member, **m**.

void break_link(
void* ptr_to_obj,

```
    void* ptr_to_obj,
    const char* class_name,
    const char* function_name
) const;
```

Use this function to maintain indexes keyed by paths containing member function calls.

ptr_to_obj is the object whose state changed, requiring an update to one or more indexes. When you call these functions, supply the same value for the first and second arguments.

class_name is the name of the class that defines the member function called in the path of the indexes to be updated.

function_name is the name of the member function itself.

Call this function before you perform an update that affects the return value of any member function appearing in an index. You must make a pair of calls (one to **break_link()** and one to **make_link()**) for each such member function affected by each data member change. If there are multiple functions affected by change, then you must call it for each function that participates in an index.

Call **break_link()** just before making the change (this removes an entry from each relevant index), and call **make_link()** just after making the change (this inserts a new entry into each relevant index, indexing the object by the new value of the relevant path). You can ensure that this happens by encapsulating these calls in a member function for setting the value of the data member.

For indexes keyed by paths that go through the elements of a collection (for example, *** ((*get_children())[]->get_location())**), index maintenance is performed automatically when you change the membership of a collection.

os_backptr::make_link()

```
void make_link(void *, void*, os_int32 = 0) const;
```

Inserts an entry into the index associated with the indexable data member pointed to by the **void*** arguments. For a class, **C**, with **os_backptr** member **C::b** and indexable member **m**, the **void*** arguments should both be

```
    &m
```

and the **os_int32** argument should be

os_index(C,b) – os_index(C,m)

The **this** pointer points to the **os_backptr** for the object being indexed. The inserted entry indexes this object by the value of the specified member, **m**.

```
void make_link(
    void* ptr_to_obj,
    void* ptr_to_obj,
    const char* class_name,
    const char* function_name
) const;
```

Use this function to maintain indexes keyed by paths containing member function calls.

ptr_to_obj is the object whose state changed, requiring an update to one or more indexes. When you call these functions, supply the same value for the first and second arguments.

class_name is the name of the class that defines the member function called in the path of the indexes to be updated.

function_name is the name of the member function itself.

Call this function before you perform an update that affects the return value of any member function appearing in an index. You must make a pair of calls (one to **break_link()** and one to **make_link()**) for each such member function affected by each data member change. If there are multiple functions affected by change, then you must call it for each function that participates in an index.

Call **break_link()** just before making the change (this removes an entry from each relevant index), and call **make_link()** just after making the change (this inserts a new entry into each relevant index, indexing the object by the new value of the relevant path). You can ensure that this happens by encapsulating these calls in a member function for setting the value of the data member.

For indexes keyed by paths that go through the elements of a collection (for example, *** ((*get_children())[]->get_location())**), index maintenance is performed automatically when you change the membership of a collection.

os_Bag

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

Characteristics

A bag (sometimes called a *multiset*) is an unordered collection. Unlike 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 one 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.

Values can be inserted into an **os_Bag** anywhere. That is, the user has no control over the ordering of the elements.

Create bags with the member **create()** or, for stack-based or embedded bags, with a constructor. Do not use **new** to create bags.

In summary, every bag has the following properties:

- Its entries have no intrinsic order.
- Duplicate elements are allowed.

Using the **behavior** argument, you can customize the behavior of new bags. See Customizing Collection Behavior in the *ObjectStore Advanced C++ API User Guide* for further information.

You can also presize a bag with a nondefault value when it is created. See **os_Bag::create()** on page 41 and **os_Bag::os_Bag()** on page 45.

Representation policy

The default representation policy for newly created bags is as follows:

- A bag created as an embedded object has the representation of **os_tiny_array** (0 to 4 elements). An embedded bag becomes out of line and mutates to an **os_chained_list** when the fifth element is inserted.
- A bag created with **::create** with cardinality <= 20 is represented as an **os_chained_list**.
- Once the bag grows past 20, **os_dyn_bag** is used, unless the array has **maintain_cursors** behavior, in which case **os_packed_list** is used.

Using the **behavior** argument, you can customize the representation of a new bag. For more information, see Customizing Collection Representation in the *ObjectStore Advanced C++ API User Guide*.

Parameterized classes

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 46). This means that when specifying **os_Bag** as a function's formal parameter, or as the type of a variable or data member, you must specify the parameter (the bag's *element type*). This is accomplished by appending to **os_Bag** the name of the element type enclosed in angle brackets, < >:

os_Bag<*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_Bag** must be a pointer type.

Required header files

Programs using bags must include the header file **<ostore/coll.hh>** after including **<ostore/ostore.hh>**.

Required libraries

Programs using bags must link with the library file **oscol.lib** (UNIX platforms) or **oscol.ldb** (Windows platforms).

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.

Tables of member functions and enumerators

Below are two tables. The first table 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*
add_index	**(const os_index_path&, os_int32, os_segment* = 0)**	**void**	**os_collection**
	(const os_index_path&, os_int32, os_database* = 0)		
	(const os_index_path&, os_segment* = 0)		
	(const os_index_path&, os_database* = 0)		
cardinality	**( ) const**	**os_int32**	**os_collection**
change_behavior	**(os_unsigned_int32 behavior_enums, os_int32 = verify_enum)**	**void**	**os_collection**
change_rep	**(os_unsigned_int32 expected_size, const os_coll_rep_descriptor *policy = 0, os_int32 retain_enum = dont_associate_policy)**	**void**	**os_collection**
clear	**( )**	**void**	**os_collection**
contains	**(const E) const**	**os_int32**	**os_Collection**
count	**(const E) const**	**os_int32**	**os_Collection**
create (static)	**(os_database *db, os_unsigned_int32 behavior_enums = 0, os_int32 expected_size = 0, const os_coll_rep_descriptor *rep_policy = 0, os_int32 retain_enum = dont_associate_policy)**	**os_Bag<E>&**	**os_Bag**
	(os_segment *seg, os_unsigned_int32 behavior_enums = 0, os_int32 expected_size = 0, const os_coll_rep_descriptor *rep_policy = 0, os_int32 retain_enum = dont_associate_policy)	**os_Bag<E>&**	
	(os_object_cluster *clust, os_unsigned_int32 behavior_enums = 0, os_int32 expected_size = 0, const os_coll_rep_descriptor *rep_policy = 0, os_int32 retain_enum = dont_associate_policy)	**os_Bag<E>&**	

Name	*Arguments*	*Returns*	*Defined By*
	(void *proximity, os_unsigned_int32 behavior_enums = 0, os_int32 expected_size = 0, const os_coll_rep_descriptor *rep_policy = 0, os_int32 retain_enum = dont_associate_policy)	os_Bag<E>&	
default_behavior (static)	( )	os_unsigned_int32	os_Bag
destroy (static)	(os_Bag<E>&)	void	os_Bag
drop_index	(const os_index_path&)	void	os_collection
empty	( )	os_int32	os_collection
exists	(char *element_type_name, char *query_string, os_database *schema_database = 0, char *file, os_unsigned_int32 line) const	os_int32	os_collection
	(const os_bound_query&) const	os_int32	
get_behavior	( ) const	os_unsigned_int32	os_collection
get_rep	( ) const	os_coll_rep_ descriptor&	os_collection
has_index	(const os_index_path&, os_int32 index_options) const	os_int32	os_collection
insert	(const E)	void	os_Collection
multi_trans_add_ index	static void multi_trans_add_index(os_reference c, const os_index_path & p, os_int32 index_options, os_segment * index_seg, os_segment * scratch_seg, os_unsigned_int32 num_per_trans)		os_collection
multi_trans_drop_ index	static void multi_trans_drop_index(os_reference c, const os_index_path & p, os_segment * scratch_seg, os_unsigned_int32 num_per_trans)	void	os_Collection
only	( ) const	E	os_Collection
operator os_Array<E>&	( )		os_Collection
operator const os_Array<E>&	( ) const		os_Collection

Name	*Arguments*	*Returns*	*Defined By*
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**	**os_int32**	**os_Collection**
	(const E) const	**os_int32**	
operator !=	**(const os_Collection<E>&) const**	**os_int32**	**os_Collection**
	(const E) const	**os_int32**	
operator <	**(const os_Collection<E>&) const**	**os_int32**	**os_Collection**
	(const E) const	**os_int32**	
operator <=	**(const os_Collection<E>&) const**	**os_int32**	**os_Collection**
	(const E) const	**os_int32**	
operator >	**(const os_Collection<E>&) const**	**os_int32**	**os_Collection**
	(const E) const	**os_int32**	
operator >=	**(const os_Collection<E>&) const**	**os_int32**	**os_Collection**
	(const E) const	**os_int32**	

Name	***Arguments***	***Returns***	***Defined By***
operator =	**(const os_Bag<E>&) const**	**os_Bag<E>&**	**os_Bag**
	(const os_Collection<E>&) const	**os_Bag<E>&**	
	(const E) const	**os_Bag<E>&**	
operator |=	**(const os_Collection<E>&) const**	**os_Bag<E>&**	**os_Bag**
	(const E) const	**os_Bag<E>&**	
operator |	**(const os_Collection<E>&) const**	**os_Collection<E>&**	**os_Collection**
	(const E) const	**os_Collection<E>&**	
operator &=	**(const os_Collection<E>&) const**	**os_Bag<E>&**	**os_Bag**
	(const E) const	**os_Bag<E>&**	
operator &	**(const os_Collection<E>&) const**	**os_Collection<E>&**	**os_Collection**
	(const E) const	**os_Collection<E>&**	
operator –=	**(const os_Collection<E>&) const**	**os_Bag<E>&**	**os_Bag**
	(const E) const	**os_Bag<E>&**	
operator -	**(const os_Collection<E>&) const**	**os_Collection<E>&**	**os_Collection**
	(const E) const	**os_Collection<E>&**	
os_Bag	**( )**		**os_Bag**
	(os_collection_size expected_size)		
	(const os_Bag<E>&)		
	(const os_Collection<E>&)		
pick	**( ) const**	**E**	**os_Collection**
	(const os_index_path&, const os_coll_range&) const	**E**	
query	**(char *element_type_name, char *query_string, os_database *schema_database = 0, char *filename, os_unsigned_int32) const**	**os_Collection<E>&**	**os_Collection**
	(const os_bound_query&) const	**os_Collection<E>&**	
query_pick	**(char *element_type_name, char *query_string, os_database *schema_database = 0) const**	**E**	**os_Collection**
	(const os_bound_query&) const	**E**	
remove	**(const E)**	**os_int32**	**os_Collection**

Name	*Arguments*	*Returns*	*Defined By*
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_Bag enumerators

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

Enumerator	*Inherited From*
allow_duplicates	**os_collection**
allow_nulls	**os_collection**
associate_policy	**os_collection**
dont_associate_policy	**os_collection**
dont_verify	**os_collection**
EQ	**os_collection**
GT	**os_collection**
LT	**os_collection**
maintain_cursors	**os_collection**
maintain_order	**os_collection**
pick_from_empty_returns_null	**os_collection**
signal_cardinality	**os_collection**
signal_duplicates	**os_collection**
unordered	**os_collection**
verify	**os_collection**

os_Bag::create()

```
static os_Bag<E> &create(
    os_database *db,
    os_unsigned_int32 behavior_enums = 0,
    os_int32 expected_size = 0,
    const os_coll_rep_descriptor *rep_policy = 0,
    os_int32 retain_enum = dont_associate_policy
);
```

Creates a new bag in the database pointed to by **db**. If the transient database is specified, the bag is allocated in transient memory.

A new bag has the following default properties:

- Its entries have no intrinsic order.
- Duplicate elements are allowed **os_collection::allow duplicates** (required)
- Performing **pick()** on an empty result of querying the bag raises err_coll_empty.
- 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.

Using the **behavior** argument, you can customize the behavior of new bags with regard to the last three properties. See Customizing Collection Behavior in the *ObjectStore Advanced C++ API User Guide*.

An **os_Bag** can have the following additional behaviors:

- **os_collection::pick_from_empty_returns_null**
- **os_collection::allow_nulls**

You can also customize the representation of a new collection (see Customizing Collection Representation in the *ObjectStore Advanced C++ API User Guide).*

The default representation policy for bags created with **create()** is as follows:

- A bag created as an embedded object has a representation of **os_tiny_array** (0 to 4 elements). An embedded bag becomes out of line when the fifth element is inserted and the representation mutates to **os_chained _list**.
- Nonembedded bags have a representation of **os_chained_list** if the expected size is less than 20.
- Once the bag grows past 20, **os_dyn_bag** is used, unless the bag has **maintain_cursors** behavior, in which case **os_packed_list** is used. (See the description in Chapter 4, Advanced Collections, of *ObjectStore Advanced C++ API User Guide*.)

If you want a new collection presized for a different cardinality, supply the **expected_size** argument explicitly.

So, for example, if **expected_size** is 21, **os_dyn_bag** is used for the collection's entire lifetime (unless you use **change_rep()**).

```
static os_Bag<E> &create(
    os_segment * seg,
    os_unsigned_int32 behavior_enums = 0,
    os_int32 expected_size = 0,
    const os_coll_rep_descriptor *rep_policy = 0,
    os_int32 retain_enum = dont_associate_policy
);
```

Creates a new bag in the segment pointed to by **seg**. If the transient segment is specified, the bag is allocated in transient memory. The rest of the arguments are just as described above.

```
static os_Bag<E> &create(
    os_object_cluster *clust,
    os_unsigned_int32 behavior = 0,
    os_int32 expected_size = 0,
    const os_coll_rep_descriptor *rep_policy = 0,
    os_int32 retain = dont_associate_policy
);
```

Creates a new bag in the object cluster pointed to by **clust**. The rest of the arguments are just as described above.

```
static os_Bag<E> &create(
    void * proximity,
    os_unsigned_int32 behavior = 0,
    os_int32 expected_size = 0,
    const os_coll_rep_descriptor *rep_policy = 0,
    os_int32 retain = dont_associate_policy
);
```

Creates a new bag in the segment occupied by the object pointed to by **proximity**. If the object is part of an object cluster, the new bag is allocated in that cluster. If the specified object is transient, the bag is allocated in transient memory. The rest of the arguments are just as described above.

os_Bag::default_behavior()

```
static os_unsigned_int32 default_behavior();
```

Returns a bit pattern indicating this type's default behavior, which is **os_collection::allow_duplicates**.

os_Bag::destroy()

```
static void destroy(os_Bag<E>&);
```

Deletes the specified collection and deallocates associated storage. This is the same as calling delete on the **os_Bag.**

Assignment Operator Semantics

Note: The assignment operator semantics are described below in terms of insert operations into the target collection. Describing the semantics in terms of insert operations serves to illustrate how duplicate, null, and order semantics are enforced. The actual implementation of the assignment might be quite different, while still maintaining the associated semantics.

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.

Note: Assignment operator semantics are described in terms of insert operations into the target bag. Note, however that the actual implementation might be different, while still maintaining the associated semantics.

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. It 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, it clears the target collection. It 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. It returns the target collection.

os_Bag::os_Bag()

os_Bag();

Returns an empty bag.

os_Bag(os_collection_size);

The user should pass an **os_int32** as the **os_collection_size** argument. 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, once 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 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 one 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 35.

Required header files

Program that use bags must include the header file **<ostore/coll.hh>** after including **<ostore/ostore.hh>**.

Required libraries

Programs that use bags must link with the library file **oscol.lib** (UNIX platforms) or **oscol.ldb** (Windows platforms).

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.

Tables of member functions and enumerators

Below are two tables. The first table 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*
add_index	**(const os_index_path&, os_int32, os_segment* = 0)**	**void**	**os_collection**
	(const os_index_path&, os_int32, os_database* = 0)	**void**	
	(const os_index_path&, os_segment* = 0)	**void**	
	(const os_index_path&, os_database* = 0)	**void**	
cardinality	**( ) const**	**os_int32**	**os_collection**
change_behavior	**(os_unsigned_int32 behavior_enums, os_int32 = verify_enum)**	**void**	**os_collection**
change_rep	**(os_unsigned_int32 expected_size, const os_coll_rep_descriptor *policy = 0, os_int32 retain_enum = dont_associate_policy)**	**void**	**os_collection**
clear	**( )**	**void**	**os_collection**
contains	**(const void*) const**	**os_int32**	**os_collection**
count	**(const void*) const**	**os_int32**	**os_collection**
create (static)	**(os_database *db, os_unsigned_int32 behavior_enums = 0, os_int32 expected_size = 0, const os_coll_rep_descriptor *rep_policy = 0, os_int32 retain_enum = dont_associate_policy)**	**os_bag&**	**os_bag**
	(os_segment *seg, os_unsigned_int32 behavior_enums = 0, os_int32 expected_size = 0, const os_coll_rep_descriptor *rep_policy = 0, os_int32 retain_enum = dont_associate_policy)	**os_bag&**	
	(os_object_cluster *clust, os_unsigned_int32 behavior_enums = 0, os_int32 expected_size = 0, const os_coll_rep_descriptor *rep_policy = 0, os_int32 retain_enum = dont_associate_policy)	**os_bag&**	

Name	*Arguments*	*Returns*	*Defined By*
	(void *proximity, os_unsigned_int32 behavior_enums = 0, os_int32 expected_size = 0, const os_coll_rep_descriptor *rep_policy = 0, os_int32 retain_enum = dont_associate_policy)	os_bag&	
default_behavior (static)	( )	os_unsigned_int32	os_bag
destroy (static)	(os_bag&)	void	os_bag
drop_index	(const os_index_path&)	void	os_collection
empty	( )	os_int32	os_collection
exists	(char *element_type_name, char *query_string, os_database *schema_database = 0, char *file, os_unsigned_int32 line) const	os_int32	os_collection
	(const os_bound_query&) const	os_int32	
get_behavior	( ) const	os_unsigned_int32	os_collection
get_rep	( ) const	os_coll_rep_ descriptor&	os_collection
has_index	(const os_index_path&, os_int32 index_options) const	os_int32	os_collection
insert	(const void*)	void	os_collection
multi_trans_add_ index	static void multi_trans_add_index(os_reference c, const os_index_path & p, os_int32 index_options, os_segment * index_seg, os_segment * scratch_seg, os_unsigned_int32 num_per_trans)		os_collection
multi_trans_drop_ index	static void multi_trans_drop_index(os_reference c, const os_index_path & p, os_segment * scratch_seg, os_unsigned_int32 num_per_trans)	void	os_Collection
only	( ) const	void*	os_Collection
operator os_array&	( )		os_collection
operator const os_array&	( ) const		os_collection

Name	*Arguments*	*Returns*	*Defined By*
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**	**os_int32**	**os_collection**
	(const void*) const	**os_int32**	
operator !=	**(const os_collection&) const**	**os_int32**	**os_collection**
	(const void*) const	**os_int32**	
operator <	**(const os_collection&) const**	**os_int32**	**os_collection**
	(const void*) const	**os_int32**	
operator <=	**(const os_collection&) const**	**os_int32**	**os_collection**
	(const void*) const	**os_int32**	
operator >	**(const os_collection&) const**	**os_int32**	**os_collection**
	(const void*) const	**os_int32**	
operator >=	**(const os_collection&) const**	**os_int32**	**os_collection**
	(const void*) const	**os_int32**	
operator =	**(const os_bag&) const**	**os_bag&**	**os_bag**
	(const os_collection&) const	**os_bag&**	
	(const void*) const	**os_bag**	
operator |=	**(const os_collection&) const**	**os_bag&**	**os_bag**
	(const void*) const	**os_bag&**	
operator |	**(const os_collection&) const**	**os_bag&**	**os_bag**
	(const void*) const	**os_bag&**	
operator &=	**(const os_collection&) const**	**os_bag&**	**os_bag**
	(const void*) const	**os_bag&**	
operator &	**(const os_collection&) const**	**os_bag&**	**os_bag**
	(const void*) const	**os_bag&**	
operator –=	**(const os_collection&) const**	**os_bag&**	**os_bag**
	(const void*) const	**os_bag&**	

Name	*Arguments*	*Returns*	*Defined By*
operator -	**(const os_collection&) const**	**os_bag&**	**os_bag**
	(const void*) const	**os_bag&**	
os_bag	**( )**		**os_bag**
	(os_collection_size expected_size)		
	(const os_bag&)		
	(const os_collection&)		
pick	**( ) const**	**void***	**os_collection**
	(const os_index_path&, const os_coll_range&) const	**void***	
query	**(char *element_type_name, char *query_string, os_database *schema_database = 0, char *filename, os_unsigned_int32) const**	**os_collection&**	**os_collection**
	(const os_bound_query&) const	**os_collection&**	
query_pick	**(char *element_type_name, char *query_string, os_database *schema_database = 0) const**	**void***	**os_collection**
	(const os_bound_query&) const	**void***	
remove	**(const void*)**	**os_int32**	**os_collection**
remove_at	**(const os_cursor&)**	**void**	**os_collection**
replace_at	**(const void*, const os_cursor&)**	**void***	**os_collection**
retrieve	**(const os_cursor&) const**	**void***	**os_collection**

os_bag enumerators

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

Name	*Inherited From*
allow_duplicates	**os_collection**
allow_nulls	**os_collection**
associate_policy	**os_collection**
dont_associate_policy	**os_collection**
dont_verify	**os_collection**
EQ	**os_collection**

Name	***Inherited From***
GT	**os_collection**
LT	**os_collection**
maintain_cursors	**os_collection**
maintain_order	**os_collection**
pick_from_empty_returns_null	**os_collection**
signal_cardinality	**os_collection**
signal_duplicates	**os_collection**
unordered	**os_collection**
verify	**os_collection**

os_bag::create()

```
static os_bag &create(
    os_database *db,
    os_unsigned_int32 behavior = 0,
    os_int32 expected_size = 0,
    const os_coll_rep_descriptor *rep_policy = 0,
    os_int32 retain = dont_associate_policy
);
```

Creates a new bag in the database pointed to by **db**. If the transient database is specified, the bag is allocated in transient memory.

Every bag has the following properties:

- Its entries have no intrinsic order.
- Duplicate elements are allowed.

By default a new bag also has the following properties:

- Performing **pick()** on an empty result of querying the bag raises **err_coll_empty**.
- 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.

Using the **behavior** argument, you can customize the behavior of new bags with regard to these last three properties. See Customizing Collection Behavior in the *ObjectStore Advanced C++ API User Guide*.

By default, bags are presized with a representation suitable for cardinality 20 or less. If you want a new collection presized for a different cardinality, supply the **expected_size** argument explicitly.

If you want to customize the representation of a new collection, see Customizing Collection Representation in the *ObjectStore Advanced C++ API User Guide*.

The default representation policy for bags is as follows:

- As the collection grows from 0 to 15, the representation is **os_chained_list**.
- Once the collection grows past 15, **os_dyn_bag** is used, unless the collection has **maintain_cursors** behavior, in which case **os_packed_list** is used.

Note that **expected_size** determines the initial representation. So, for example, if **expected_size** is 21, **os_dyn_bag** is used for the collection's entire lifetime (unless you use **change_rep()**).

```
static os_bag &create(
    os_segment * seg,
    os_unsigned_int32 behavior = 0,
    os_int32 expected_size = 0,
    const os_coll_rep_descriptor *rep_policy = 0,
    os_int32 retain = dont_associate_policy
);
```

Creates a new bag in the segment pointed to by **seg**. If the transient segment is specified, the bag is allocated in transient memory. The rest of the arguments are just as described above.

```
static os_bag &create(
    os_object_cluster *clust,
    os_unsigned_int32 behavior = 0,
    os_int32 expected_size = 0,
    const os_coll_rep_descriptor *rep_policy = 0,
    os_int32 retain = dont_associate_policy
);
```

Creates a new bag in the object cluster pointed to by **clust**. The rest of the arguments are just as described above.

```
static os_bag &create(
    void * proximity,
    os_unsigned_int32 behavior = 0,
    os_int32 expected_size = 0,
    const os_coll_rep_descriptor *rep_policy = 0,
```

```
    os_int32 retain = dont_associate_policy
);
```

Creates a new bag in the segment occupied by the object pointed to by **proximity**. If the object is part of an object cluster, the new collection is allocated in that cluster. If the specified object is transient, the bag is allocated in transient memory. The rest of the arguments are just as described above.

os_bag::default_behavior()

static os_unsigned_int32 default_behavior();

Returns a bit pattern indicating this type's default behavior.

os_bag::destroy()

static void destroy(os_bag&);

Deletes the specified collection and deallocates associated storage.

Note: The assignment operator semantics are described below in terms of insert operations into the target collection. Describing the semantics in terms of insert operations serves to illustrate how duplicate, null, and order semantics are enforced. The actual implementation of the assignment might be quite different, while still maintaining the associated semantics.

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) 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 nulls, the result does. The result allows duplicates and does not maintain order, maintain cursors, or signal duplicates.

os_bag &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 nulls, the result does. The result allows duplicates and does not maintain order, maintain cursors, or signal duplicates.

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. It 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. It returns the target collection.

os_bag::operator &()

os_bag &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 nulls, the result does. The result allows duplicates and does not maintain order, maintain cursors, or signal duplicates.

os_bag &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 nulls, the result does. The result allows duplicates and does not maintain order, maintain cursors, or signal duplicates.

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. It 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::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 nulls, the result does. The result allows duplicates and does not maintain order, maintain cursors, or signal duplicates.

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 nulls, the result does. The result allows duplicates and does not maintain order, maintain cursors, or signal duplicates.

os_bag::os_bag()

os_bag();

Returns an empty bag.

os_bag(os_collection_size);

The user should pass an **os_int32** as the **os_collection_size** argument. 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, once 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_bound_query

Instances of this class are query objects built from instances of **os_coll_query** and **os_keyword_arg_list**. Bound queries must be transiently allocated; they should not be allocated in persistent memory.

os_bound_query::os_bound_query()

```
os_bound_query(
    const os_coll_query&,
    const os_keyword_arg_list&
);
```

Creates a bound query, binding the free references in the **os_coll_query** according to the **os_keyword_arg_list**.

```
os_bound_query(
    const os_coll_query&
);
```

Creates a bound query from an **os_coll_query** with no free references.

os_Collection

template <class E>
class os_Collection : public 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 87). 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 parameter (the collection's *element type*). 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.

Create collections with the member **create()** or, for stack-based or embedded collections, with a constructor. Do not use **new** to create collections.

Required header files

Programs that use **os_Collection**s must include the header file **<ostore/coll.hh>** after including **<ostore/ostore.hh>**.

Required libraries

Programs that use **os_Collection**s must link with the library file **oscol.lib** (UNIX platforms) or **oscol.ldb** (Windows platforms).

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.

Below are two tables. The first table 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*
add_index	**(const os_index_path&, os_int32 = unordered, os_segment* = 0)**	**void**	**os_collection**
	(const os_index_path&, os_int32 = unordered, os_database* = 0)	**void**	
	(const os_index_path&, os_segment* = 0)	**void**	
	(const os_index_path&, os_database* = 0)	**void**	
cardinality	**( ) const**	**os_unsigned_int32**	**os_collection**
change_behavior	**(os_unsigned_int32 behavior, os_int32 = verify)**	**void**	**os_collection**
change_rep	**(os_unsigned_int32 expected_size const os_coll_rep_descriptor *policy = 0, os_int32 retain = dont_associate_policy)**	**void**	**os_collection**
clear	**( )**	**void**	**os_collection**
contains	**(const E) const**	**os_int32**	**os_Collection**
count	**(const E) const**	**os_int32**	**os_Collection**
create (static)	**(os_segment *seg, os_unsigned_int32 behavior = 0, os_int32 expected_size = 0, const os_coll_rep_descriptor* = 0, os_int32 retain = dont_associate_policy)**	**os_Collection<E>&**	**os_Collection**
	(os_database *db, os_unsigned_int32 behavior = 0, os_int32 expected_size = 0, const os_coll_rep_descriptor* = 0, os_int32 retain = dont_associate_policy)	**os_Collection<E>&**	

Name	***Arguments***	***Returns***	***Defined By***
	(os_object_cluster *cluster, os_unsigned_int32 behavior = 0, os_int32 expected_size = 0, const os_coll_rep_descriptor* = 0, os_int32 retain = dont_associate_policy)	**os_Collection<E>&**	
	(void *proximity, os_unsigned_int32 behavior = 0, os_int32 expected_size = 0, const os_coll_rep_descriptor* = 0, os_int32 retain = dont_associate_policy)	**os_Collection<E>&**	
default_behavior (static)	**( )**	**os_unsigned_int32**	**os_collection**
destroy (static)	**(os_Collection<E>&)**	**void**	**os_Collection**
drop_index	**(const os_index_path&)**	**void**	**os_collection**
empty	**( )**	**os_int32**	**os_collection**
exists	**(char *element_type_name, char *query_string, os_database *schema_database = 0, char *file, os_unsigned_int32 line) const**	**os_int32**	**os_collection**
	(const os_bound_query&) const	**os_int32**	
get_behavior	**( ) const**	**os_unsigned_int32**	**os_collection**
get_rep	**( ) const**	**os_coll_rep_ descriptor&**	**os_collection**
has_index	**(const os_index_path&, os_int32 index_options = unordered) const**	**os_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**

Name	*Arguments*	*Returns*	*Defined By*
multi_trans_add_ index	**(os_reference c, const os_index_path & p, os_int32 index_options, os_segment * index_seg, os_segment * scratch_seg, os_unsigned_int32 num_per_trans)**		**os_collection**
multi_trans_drop_ index	**(os_reference c, const os_index_path & p, os_segment * scratch_seg, os_unsigned_int32 num_per_trans)**		**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**

Name	*Arguments*	*Returns*	*Defined By*
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**	
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**	
operator =	**(const os_Collection<E>&) const**	**os_Collection<E>&**	**os_Collection**
	(E) const	**os_Collection<E>&**	
operator |=	**(const os_Collection<E>&) const**	**os_Collection<E>&**	**os_Collection**
	(E) const	**os_Collection<E>&**	
operator |	**(const os_Collection<E>&) const**	**os_Collection<E>&**	**os_Collection**
	(E) const	**os_Collection<E>&**	
operator &=	**(const os_Collection<E>&) const**	**os_Collection<E>&**	**os_Collection**
	(E) const	**os_Collection<E>&**	
operator &	**(const os_Collection<E>&) const**	**os_Collection<E>&**	**os_Collection**
	(E) const	**os_Collection<E>&**	
operator –=	**(const os_Collection<E>&) const**	**os_Collection<E>&**	**os_Collection**
	(E) const	**os_Collection<E>&**	
operator -	**(const os_Collection<E>&) const**	**os_Collection<E>&**	**os_Collection**
	(E) const	**os_Collection<E>&**	
pick	**( ) const**	**E**	**os_Collection**
	(const os_index_path&, const os_coll_range&) const	**E**	

Name	*Arguments*	*Returns*	*Defined By*
query	**(char *element_type_name, char *query_string, os_database *schema_database = 0, char *file, os_unsigned_int32 line, os_boolean dups) const**	**os_Collection<E>&**	**os_Collection**
	(const os_bound_query&) const	**os_Collection<E>&**	
query_pick	**(char *element_type_name, char *query_string, os_database *schema_database = 0, char *file, os_unsigned_int32 line) const**	**E**	**os_Collection**
	(const os_bound_query&) const	**E**	
remove	**(const E)**	**os_int32**	**os_Collection**
remove_at	**(const os_Cursor<E>&)**	**void**	**os_Collection**
	(os_unsigned_int32)	**void**	
remove_first	**(const E&)**	**os_int32**	**os_Collection**
	( )	**E**	
remove_last	**(const E&)**	**os_int32**	**os_Collection**
	( )	**E**	
replace_at	**(const E, const os_Cursor<E>&)**	**E**	**os_Collection**
	(E, os_unsigned_int32)	**E**	
retrieve	**(os_unsigned_int32) const**	**E**	**os_Collection**
	(const os_Cursor<E>&) const	**E**	
retrieve_first	**( ) const**	**E**	**os_Collection**
	(E&) const	**os_int32**	
retrieve_last	**( ) const**	**E**	**os_Collection**
	(E&) const	**os_int32**	

os_Collection enumerators

The following table lists the enumerators for **os_Collection**.

Name	*Inherited From*
allow_duplicates	**os_collection**
allow_nulls	**os_collection**
associate_policy	**os_collection**

Name	*Inherited From*
dont_associate_policy	**os_collection**
dont_verify	**os_collection**
EQ	**os_collection**
GT	**os_collection**
LT	**os_collection**
maintain_cursors	**os_collection**
maintain_order	**os_collection**
pick_from_empty_returns_null	**os_collection**
signal_cardinality	**os_collection**
signal_duplicates	**os_collection**
unordered	**os_collection**
verify	**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::create()

```
static os_Collection<E> &create(
    os_database *db,
    os_unsigned_int32 behavior = 0,
    os_int32 expected_size = 0,
    const os_coll_rep_descriptor *rep_policy = 0,
    os_int32 retain = dont_associate_policy
);
```

Creates a new collection in the database pointed to by **db**. If the transient database is specified, the collection is allocated in transient memory.

Every instance of **os_Collection** has the following properties:

- Its entries have no intrinsic order.
- Duplicate elements are not allowed.

By default a new **os_Collection** object also has the following properties:

- Performing **pick()** on an empty result of querying the collection raises err_coll_empty.
- 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.

Using the **behavior** argument, you can customize the behavior of new **os_Collection**s with regard to these last three properties. See Customizing Collection Behavior in the *ObjectStore Advanced C++ API User Guide*.

Collections are the most flexible container class. The behaviors that they can have are

- **allow_duplicates**
- **maintain_order**
- **signal_duplicates**
- **allow_nulls**
- **pick_from_empty_returns_null**
- **maintain_cursors**

By default, instances of **os_Collection** are presized with a representation suitable for cardinality 20 or less. If you want a new collection presized for a different cardinality, supply the **expected_size** argument explicitly.

If you want to customize the representation of a new collection, see Customizing Collection Representation in the *ObjectStore Advanced C++ API User Guide*.

The default representation policy for **os_Collection**s is as follows:

- A collection created as an embedded object has the representation of **os_tiny_array** (0 to 4 elements).
- An embedded collection becomes out of line and mutates to an **os_chained_list** when the fifth element is inserted.

- A collection created with **::create** with cardinality <= 20 is represented as an **os_chained_list**.
- Once the collection grows past 20, **os_dyn_hash** is used for the collection's entire lifetime (unless you use **change_rep**).

Note that **expected_size** determines the initial representation. So, for example, if **expected_size** is 21, **os_dyn_hash** is used for the collection's entire lifetime (unless you use **change_rep()**.

```
static os_Collection<E> &create(
    os_segment * seg,
    os_unsigned_int32 behavior = 0,
    os_int32 expected_size = 0,
    const os_coll_rep_descriptor *rep_policy = 0,
    os_int32 retain = dont_associate_policy
);
```

Creates a new collection in the segment pointed to by **seg**. If the transient segment is specified, the collection is allocated in transient memory. The rest of the arguments are just as described above.

```
static os_Collection<E> &create(
    os_object_cluster *clust,
    os_unsigned_int32 behavior = 0,
    os_int32 expected_size = 0,
    const os_coll_rep_descriptor *rep_policy = 0,
    os_int32 retain = dont_associate_policy
);
```

Creates a new collection in the object cluster pointed to by **clust**. The rest of the arguments are just as described above.

```
static os_Collection<E> &create(
    void * proximity,
    os_unsigned_int32 behavior = 0,
    os_int32 expected_size = 0,
    const os_coll_rep_descriptor *rep_policy = 0,
    os_int32 retain = dont_associate_policy
);
```

Creates a new collection in the segment occupied by the object pointed to by **proximity**. If the object is part of an object cluster, the new collection is allocated in that cluster. If the specified object is transient, the collection is allocated in transient memory. The rest of the arguments are just as described above.

os_Collection::default_behavior()

os_unsigned_int32 default_behavior();

Returns a bit pattern indicating this type's default behavior. In this case, the bit pattern is 0.

os_Collection::destroy()

static void destroy(os_Collection<E>&);

Deletes the specified collection and deallocates associated storage. This is the same as calling **delete()** on an **os_Collection**.

os_Collection::drop_index()

void drop_index(const os_index_path &p);

Destroys the index into the specified collection whose key is specified by **p**. The argument **p** does *not* need to be the same instance of **os_index_path** supplied when the index was added, but it must specify the same key. Two **os_index_path**s created with the same path string and type string specify the same index key.

Collections with large cardinality might warrant removing the index with multiple transactions. See **os_collection::multi_trans_drop_index()** on page 111.

An err_no_such_index exception is signaled if an index with the specified key does not exist for the collection.

os_Collection::exists()

exists(query_string);

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 duplicates (has behavior **signal_duplicates)**, and the specified **E** is already present in the

collection, err_coll_duplicates is signaled. Otherwise the insertion is ignored.

- If the collection disallows nulls, and the specified **E** is **0**, err_coll_nulls is signaled.

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_null_cursor is signaled. If the cursor is invalid, err_coll_illegal_cursor is signaled.

The behavior of **insert_after()** 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 (has behavior **signal_duplicates)**, and the specified **E** is already present in the collection, err_coll_duplicates is signaled. Otherwise the insertion is ignored.
- If the collection disallows nulls, and the specified **E** is **0**, err_coll_nulls is signaled.

If the collection has behavior **maintain_cursors** and the cursor has behavior **update_safe**, the next element in the forward iteration will be **E.**

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()** 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 (has the behavior **signal_duplicates**), and the specified **E** is already present in the collection, err_coll_duplicates is signaled.
- If the collection disallows nulls, and the specified **E** is **0**, err_coll_nulls is signaled.
- If the collection has behavior **maintain_cursors** and the cursor has behavior **update_safe**, the next element in the forward iteration will be **E.**

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_null_cursor is signaled. If the cursor is invalid, err_coll_illegal_cursor is signaled.

The behavior of **insert_before()** 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 (has the behavior **signal_duplicates**), and the specified **E** is already present in the collection, err_coll_duplicates is signaled.
- If the collection disallows nulls, and the specified **E** is **0**, err_coll_nulls is signaled.
- If the collection has behavior **maintain_cursors**, and the cursor has behavior **update_safe**, the previous element in the backwards iteration will be **E**.
- If the collection is an array, all elements after this element will be pushed down.

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()** 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 (has the behavior **signal_duplicates**), and the specified **E** is already present in the collection, err_coll_duplicates is signaled.
- If the collection disallows nulls, and the specified **E** is **0**, err_coll_nulls is signaled.
- If the collection has behavior **maintain_cursors**, and the cursor has behavior **update_safe**, the previous element in the backwards iteration will be **E**.
- If the collection is an array, all elements after this element will be pushed down.

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 duplicates (has the behavior **signal_duplicates**), and the specified **E** is already present in the collection, err_coll_duplicates 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 will be pushed down.

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 duplicates (has the behavior **signal_duplicates**), and the specified **E** is already present in the collection, err_coll_duplicates is signaled.
- If the collection disallows nulls, and the specified **E** is **0**, err_coll_nulls is signaled.

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, err_coll_empty is signaled, unless the collection's behavior includes **os_collection::pick_from_empty_returns_null**, in which case **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

The assignment operator semantics are described below in terms of insert operations into the target collection. Describing the semantics in terms of insert operations serves to illustrate how duplicate, null, and order semantics are enforced. The actual implementation of the assignment might be quite different, while still maintaining the associated semantics.

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. The result does not maintain cursors or signal duplicates.

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. The result does not maintain cursors or signal duplicates.

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. It 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. 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. The result does not maintain cursors or signal duplicates.

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. The result does not maintain cursors or signal duplicates.

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. The result does not maintain cursors or signal duplicates.

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. The result does not maintain cursors or signal duplicates.

os_Collection::pick()

E pick() const;

Returns an arbitrary element of the specified collection. If the collection is empty, err_coll_empty is signaled, unless the collection's behavior includes **os_collection::pick_from_empty_returns_null**, in which case **0** is returned.

os_Collection::query()

```
os_Collection<E> &query(
    char *element_type_name,
    char *query_string,
    os_database *schema_database = 0,
    char *file_name = 0,
    os_unsigned_int32 line = 0,
```

```
    os_boolean dups = query_dont_preserve_duplicates
) const;
```

Returns a reference to a heap-allocated collection with default behavior containing those elements of **this** that satisfy the selection criterion expressed by the **query_string**. When you no longer need the resulting collection, you should reclaim its memory with **::operator delete()** or **os_collection::destroy()** to avoid memory leaks.

The argument **element_type_name** is the name of the element type of **this**. Names established through the use of **typedef** are not allowed.

The **query_string** is a C++ *control expression* indicating the query's selection criterion. An element, **e**, satisfies the selection criterion if the control expression evaluates to a nonzero **os_int32** (true) when **e** is bound to **this**.

Any string consisting of an **os_int32**-valued C++ expression is allowed in a query string, as long as

- Variables are also data members of the elements of the collection.
- For local variables (free references), you create an **os_coll_query** object.
- For global functions (free references), you create an **os_coll_query** object.
- There are no function calls, except calls to **strcmp()** or **strcoll()**.
- There are no comparison operators for which the user might be required to define a corresponding rank / hash function.
- There are no calls to member functions that satisfy the restrictions listed below.

Within the selection criterion of query expressions, member names are implicitly qualified by **this**, just as are member names in function member bodies.

Restrictions

Functions called in query strings are subject to certain restrictions:

- The return type can be a basic type (**int**, **char**, **float**, **char***).
- If the function is a member function it can also return a pointer or a reference to a class type.

- The function can take up to two arguments. The first argument must be a pointer. For member functions **this** is the implied first argument.
- Global functions are free references and must be used in an **os_coll_query** object.
- Member functions can be used like data members.

To perform a query, ObjectStore sometimes (depending on what indexes are present) issues calls to member functions used in paths and queries. If such a member function allocates memory it does not free (for example if it returns a pointer to newly allocated memory), memory leaks can result; ObjectStore does not free the space the function allocates. So member functions used in paths or queries should not allocate persistent memory or memory in the transient heap.

Member function in a query string

Applications that use a member function (*not* returning a reference) in a query string must do four things:

- Define an **os_backptr**-valued data member in the class that defines the member function. It must precede the member function declaration in the class definition.
- Call the macro **os_query_function()**. This should be defined at file scope, for example, in the header file that contains the class that defines the member function. See **os_query_function()** in Chapter 4, Macros and User-Defined Functions of the *ObjectStore Collections C++ API Reference* for more information.
- Call the macro **os_query_function_body()**. This should be defined at file scope in a source file that will only be compiled into the application once. See **os_query_function_body()** for more information.
- Call the macro **OS_MARK_QUERY_FUNCTION()**. This macro should be invoked in the schema source file. See **OS_MARK_QUERY_FUNCTION()** on page 268 for more information.

Member function, returning a reference, in a query string

For applications that use a member function that returns a reference in a query string, you must do the following four things:

- Define an **os_backptr**-valued data member in the class that defines the member function.
- Call the macro **os_query_function_returning_ref()**. This should be defined at file scope, for example, in the header file that

contains the class that defines the member function. See **os_query_function_body_returning_ref()** for more information.

- Call the macro **os_query_function_body_returning_ref()**. This should be defined at file scope in a source file that will only be compiled into the application once. See **os_query_function_body_returning_ref()** for more information.
- Call the macro **OS_MARK_QUERY_FUNCTION()**. This macro should be invoked in the schema source file. See **OS_MARK_QUERY_FUNCTION()** on page 268 for more information.

To maintain indexes keyed by paths containing member function calls, use **os_backptr::make_link()** and **os_backptr::break_link()**.

The query string can itself contain queries. A notation is defined to allow the user to conveniently specify such nested queries in a single call to a query member function.

A nested collection-valued query has the form

collection-expression **[:** *os_int32-expression* **:]**

where *collection-expression* is an expression of type **os_Collection**, and *os_int32-expression* is the selection criterion for the nested query.

A nested single-element query has the form

collection-expression **[%** *os_int32-expression* **%]**

where *collection-expression* and *os_int32-expression* are as for nested collection-valued queries. This form evaluates to one element of *collection-expression*. If there is more than one element that satisfies the nested query's selection criterion, one of them is picked and returned.

A nested query returning a collection will be converted to an **os_int32** when appropriate, using **os_collection::operator os_int32()**.

The **schema_database** is a database whose schema contains all the types mentioned in the selection criterion. This database provides the environment in which the query is analyzed and optimized. If this argument is not supplied, the database in which the collection resides is used, which is always adequate. If the query is being performed over a transient collection, the application schema database is used by default.

In addition to being used as the default schema database for queries over transient collections, the application schema database is used if the transient database is supplied explicitly.

Since the application schema database is never opened just prior to analysis of a query, use of the application schema database in query analysis carries the overhead of a database open. Therefore, explicitly supplying an appropriate database that is already open will improve query performance.

ObjectStore uses **file_name** and **line** when reporting errors related to the query. You can set them to identify the location of the query's source code.

If **dups** is the enumerator **os_collection::query_dont_preserve_duplicates**, duplicate elements that satisfy the query condition are not included in the query result. If **dups** is the enumerator **os_collection::query_preserve_duplicates**, duplicate elements that satisfy the query condition are included in the query result.

```
os_Collection <E> &query(
    const os_bound_query&,
    os_boolean dups = query_dont_preserve_duplicates
) const;
```

Returns a reference to a heap-allocated collection with default behavior containing those elements of **this** that satisfy the **os_bound_query**. If **dups** is the enumerator **query_dont_preserve_duplicates**, duplicate elements that satisfy the query condition are not included in the query result. If **dups** is the enumerator **query_preserve_duplicates**, duplicate elements that satisfy the query condition are included in the query result.

When you no longer need the resulting collection, you should reclaim its memory with **::operator delete()** or **os_collection::destroy()** to avoid memory leaks.

os_Collection::query_pick()

```
E query_pick(
    char *element_type_name,
    char *query_string,
    os_database *schema_database = 0,
    char *filename, os_unsigned_int32 line,
) const;
```

Returns an element of **this** that satisfies the selection criterion expressed by the **query_string**. If there is more than one such element, one is picked arbitrarily and returned. If no element satisfies the query or the collection is empty, **0** is returned.

The argument **element_type_name** is the name of the element type of **this**. Names established through the use of **typedef** are not allowed.

The **query_string** is a C++ *control expression* indicating the query's selection criterion. An element, **e**, satisfies the selection criterion if the control expression evaluates to a nonzero **os_int32** (true) when **e** is bound to **this**.

Any string consisting of an **os_int32**-valued C++ expression is allowed, as long as

- Variables are also data members of the elements of the collection.
- For local variables (free references), you create an **os_coll_query** object.
- For global functions (free references), you create an **os_coll_query** object.
- There are no function calls, except calls to **strcmp()** or **strcoll()**.
- There are no comparison operators for which the user might be required to define a corresponding rank / hash function.
- There are no calls to member functions that satisfy the restrictions listed below.

Within the selection criterion of query expressions, member names are implicitly qualified by **this**, just as are member names in function member bodies.

Restrictions

Functions called in query strings are subject to certain restrictions:

- The return type can be a basic type (**int**, **char**, **float**, **char***).
- If the function is a member function it can also return a pointer or a reference to a class type.
- The function can take up to two arguments. The first argument must be a pointer. For member functions **this** is the implied first argument.

- Global functions are free references and must be used in an **os_coll_query** object.
- Member functions can be used like data members.

To perform a query, ObjectStore sometimes (depending on what indexes are present) issues calls to member functions used in paths and queries. If such a member function allocates memory it does not free (for example if it returns a pointer to newly allocated memory), memory leaks can result; ObjectStore does not free the space the function allocates. So member functions used in paths or queries should not allocate persistent memory or memory in the transient heap.

Member function in a query string

Applications that use a member function (*not* returning a reference) in a query string must do four things:

- Define an **os_backptr**-valued data member in the class that defines the member function. It must precede the member function declaration in the class definition.
- Call the macro **os_query_function()**. This should be defined at file scope, for example, in the header file that contains the class that defines the member function. See **os_query_function()** for more information.
- Call the macro **os_query_function_body()**. This should be defined at file scope in a source file that will only be compiled into the application once. See **os_query_function_body()** for more information.
- Call the macro **OS_MARK_QUERY_FUNCTION()**. This macro should be invoked in the schema source file. See **OS_MARK_QUERY_FUNCTION()** on page 268 for more information.

Member function, returning a reference, in a query string

For applications that use a member function that returns a reference in a query string, you must do the following four things:

- Define an **os_backptr**-valued data member in the class that defines the member function.
- Call the macro **os_query_function_returning_ref()**. This should be defined at file scope, for example, in the header file that contains the class that defines the member function. See **os_query_function_body_returning_ref()** for more information.
- Call the macro **os_query_function_body_returning_ref()**. This should be defined at file scope in a source file that will only be

compiled into the application once. See **os_query_function_body_returning_ref()** for more information.

- Call the macro **OS_MARK_QUERY_FUNCTION()**. This macro should be invoked in the schema source file. See **OS_MARK_QUERY_FUNCTION()** on page 268 for more information.

To maintain indexes keyed by paths containing member function calls, use **os_backptr::make_link()** and **os_backptr::break_link()**.

The query string can itself contain queries. A notation is defined to allow the user to conveniently specify such nested queries in a single call to a query member function.

A nested collection-valued query has the form

collection-expression **[:** *os_int32-expression* **:]**

where *collection-expression* is an expression of type **os_Collection**, and *os_int32-expression* is the selection criterion for the nested query.

A nested single-element query has the form

collection-expression **[%** *os_int32-expression* **%]**

where *collection-expression* and *os_int32-expression* are as for nested collection-valued queries. This form evaluates to one element of *collection-expression*. If there is more than one element that satisfies the nested query's selection criterion, one of them is picked and returned.

A nested query returning a collection is converted to an **os_int32** when appropriate, using **os_collection::operator os_int32()**.

The **schema_database** is a database whose schema contains all the types mentioned in the selection criterion. This database provides the environment in which the query is analyzed and optimized. The database in which the collection resides is often an appropriate choice for this.

If the transient database is specified, the application's schema (stored in the application schema database) is used to evaluate the query.

E query_pick(const os_bound_query&) const;

Returns an element of **this** that satisfies the **os_bound_query**. If there is more than one such element, one is picked arbitrarily and returned. If no element satisfies the query or the collection is empty, **0** is returned.

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(E&);

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 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 this 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 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 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 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()

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 with 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 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()

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.

This class has a parameterized subtype. See **os_Collection** on page 58.

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

Create collections with the member **create()** or, for stack-based or embedded collections, with a constructor. Do not use **new** to create collections.

Required include files

Any program using collections must include the header file **<ostore/coll.hh>** after including **<ostore/ostore.hh>**.

Required libraries

Programs that use **os_collection**s must link with the library file **oscol.lib** (UNIX platforms) or **oscol.ldb** (Windows platforms).

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.

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

Below are two tables. The first table 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
add_index	(const os_index_path&, os_int32 = unordered, os_segment* = 0)	void
	(const os_index_path&, os_int32 = unordered, os_database* = 0)	void
	(const os_index_path&, os_segment* = 0)	void
	(const os_index_path&, os_database* = 0)	void
cardinality	( ) const	os_unsigned_int32
change_behavior	(os_unsigned_int32 behavior, os_int32 = verify)	void
change_rep	(os_unsigned_int32 expected_size const os_coll_rep_descriptor *policy= 0, os_int32 retain = dont_associate_policy)	void
clear	( )	void
contains	(const void*) const	
count	(const void*) const	os_int32
create (static)	(os_segment *seg, os_unsigned_int32 behavior = 0, os_int32 expected_size = 0, const os_coll_rep_descriptor* = 0, os_int32 retain = dont_associate_policy)	os_collection&
	(os_database *db, os_unsigned_int32 behavior = 0, os_int32 expected_size = 0, const os_coll_rep_descriptor* = 0, os_int32 retain = dont_associate_policy)	os_collection&
	(os_object_cluster *clust, os_unsigned_int32 behavior = 0, os_int32 expected_size = 0, const os_coll_rep_descriptor* = 0, os_int32 retain = dont_associate_policy)	os_collection&
	(void* proximity, os_unsigned_int32 behavior = 0, os_int32 expected_size = 0, const os_coll_rep_descriptor* = 0, os_int32 retain = dont_associate_policy)	os_collection&
default_behavior (static)	( )	os_unsigned_int32

Name	*Arguments*	*Returns*
destroy (static)	**(os_collection&)**	**void**
drop_index	**(const os_index_path&)**	**void**
empty	**( )**	**os_int32**
exists	**(char *element_type_name, char *query_string, os_database *schema_database = 0, char *file, os_unsigned_int32 line) const**	**os_int32**
	(const os_bound_query&) const	**os_int32**
get_behavior	**( ) const**	**os_unsigned_int32**
get_indexes	**( ) const**	**os_collection***
get_rep	**( ) const**	**os_coll_rep_ descriptor&**
get_thread_locking (static)	**( )**	**os_boolean**
has_index	**(const os_index_path&, os_int32 index_options = unordered) const**	**os_int32**
insert	**(const void*)**	**void**
insert_after	**(const void*, const os_cursor&)**	**void**
	(const void*, os_unsigned_int32)	**void**
insert_before	**(const void*, const os_cursor&)**	**void**
	(const void*, os_unsigned_int32)	**void**
insert_first	**(const void*)**	**void**
insert_last	**(const void*)**	**void**
multi_trans_add_index	**(os_reference c, const os_index_path & p, os_int32 index_options, os_segment * index_seg, os_segment * scratch_seg, os_unsigned_int32 num_per_trans)**	
multi_trans_drop_index	**(os_reference c, const os_index_path & p, os_segment * scratch_seg, os_unsigned_int32 num_per_trans)**	
only	**( ) const**	**E**

Name	***Arguments***	***Returns***
operator os_array&	**( )**	
operator const os_array&	**( ) const**	
operator os_bag&	**( )**	
operator const os_bag&	**( ) const**	
operator os_list&	**( )**	
operator const os_list&	**( ) const**	
operator os_set&	**( )**	
operator const os_set&	**( ) const**	
operator ==	**(const os_collection&) const**	**os_int32**
	(const void*) const	**os_int32**
operator !=	**(const os_collection&) const**	**os_int32**
	(const void*) const	**os_int32**
operator <	**(const os_collection&) const**	**os_int32**
	(const void*) const	**os_int32**
operator <=	**(const os_collection&) const**	**os_int32**
	(const void*) const	**os_int32**
operator >	**(const os_collection&) const**	**os_int32**
	(const void*) const	**os_int32**
operator >=	**(const os_collection&) const**	**os_int32**
	(const void*) const	**os_int32**
operator =	**(const os_collection&) const**	**os_collection&**
	(const void*) const	**os_collection&**
operator |=	**(const os_collection&) const**	**os_collection&**
	(const void*) const	**os_collection&**
operator |	**(const os_collection&) const**	**os_collection&**
	(const void*) const	**os_collection&**
operator &=	**(const os_collection&) const**	**os_collection&**
	(const void*) const	**os_collection&**
operator &	**(const os_collection&) const**	**os_collection&**
	(const void*) const	**os_collection&**
operator –=	**(const os_collection&) const**	**os_collection&**
	(const void*) const	**os_collection&**

Name	***Arguments***	***Returns***
operator -	**(const os_collection&) const**	**os_collection&**
	(const void*) const	**os_collection&**
pick	**( ) const**	**void***
	(const os_index_path &path, const os_coll_range &range) const	**void***
query	**(char *element_type_name, char *query_string, os_database *schema_database = 0, char *file, os_unsigned_int32 line, os_boolean dups) const**	**os_collection&**
	(const os_bound_query&) const	**os_collection&**
query_pick	**(char *element_type_name, char *query_string, os_database *schema_database = 0, char *file, os_unsigned_int32 line) const**	**void***
	(const os_bound_query&) const	**void***
remove	**(const void*)**	**os_int32**
remove_at	**(const os_cursor&)**	**void**
	(os_unsigned_int32)	**void**
remove_first	**(const void*&)**	**os_int32**
	( )	**void***
remove_last	**(const void*&)**	**os_int32**
	( )	**void***
replace_at	**(const void*, const os_cursor&)**	**void***
	(const void*, os_unsigned_int32)	**void***
retrieve	**(os_unsigned_int32) const**	**void***
	(const os_cursor&) const	**void***
retrieve_first	**( ) const**	**void***
	(const void*&) const	**os_int32**
retrieve_last	**( ) const**	**void***
	(const void*&) const	**os_int32**
set_thread_locking (static)	**(os_boolean)**	**void**

os_collection enumerators

The following table lists enumerators for **os_collection**.

Enumerators
allow_duplicates
allow_nulls
associate_policy
dont_associate_policy
dont_verify
EQ
GE
GT
LE
LT
maintain_cursors
maintain_order
NE
order_by_address
pick_from_empty_returns_null
signal_cardinality
signal_duplicates
unordered
verify

os_collection::add_index()

```
void add_index(
    const os_index_path&,
    os_int32 options,
    os_segment* = 0
);
```

Creates an index into the specified collection, keyed by the specified **os_index_path**. The presence of the index allows optimization of queries involving lookup of collection elements based on the specified path. See the class **os_index_path** on page 179. If the specified collection already has an index with the specified key, this call is ignored. Two instances of **os_index_path** specify the same key if they were created with the same path string and element type.

Collections with large cardinality might warrant adding the index using multiple transactions. See **os_collection::multi_trans_add_index()** on page 110.

The exception err_am is signaled if a class mentioned in the path serving as index key cannot be found in the schema of the database containing the index (or the application schema, if the index is transient).

The **options** argument is a bit pattern indicating the behavior of the index. You supply the bit pattern by forming the bit-wise disjunction (using **|**) of enumerators signifying the desired behavior. These enumerators, together with the behaviors they signify, are listed below.

- **os_index_path::ordered**: indicates an ordered index, implemented as a B-tree, supporting optimization of range queries, that is, queries involving the comparison operators **<**, **>**, **<=**, and **>=**. Specifying both **ordered** and **unordered** (see below) for the same index results in an ordered index.
- **os_index_path::unordered**: indicates an unordered index, implemented as a hash table. Such an index does not support optimization of range queries. Specifying both **ordered** and **unordered** for the same index results in an ordered index.
- **os_index_path::allow_duplicates**: indicates an index that allows duplicate keys. You should use such an index for collections in which two or more elements can share a key value. Specifying both **allow_duplicates** and **no_duplicates** (see below) for the same index results in a **no_duplicates** index.
- **os_index_path::no_duplicates**: indicates an index that does not allow duplicate key values. You should use such an index for collections in which no two elements can share a key value. If duplicate key values might accidentally occur, use this enumerator together with **os_index_path::signal_duplicates** (see below). Without **signal_duplicates**, duplicate keys will not be detected and can have unpredictable results. Specifying both **allow_duplicates** and **no_duplicates** (see above) for the same index results in a **no_duplicates** index.
- **os_index_path::signal_duplicates**: indicates an index that detects duplicate key values. Can only be used together with **os_index_path::no_duplicates**. If an index that signals

duplicates is added to a collection containing two or more elements that share a key value, the exception err_index_duplicate_key is signaled. In addition, for a collection with an index that signals duplicates, inserting an element with the same key value as some other element also provokes an err_index_duplicate_key exception.

- **os_index_path::copy_key** (default): indicates an index with entries consisting of key-value / element pairs, as opposed to pointer-to-key-value / element pairs (see **os_index_path::point_to_key**, below). For a **copy_key** index, form an entry by copying the object at the end of the **os_index_path** that specifies the key. Such an index generally takes up more space than one that points to its keys, but it provides faster access times because of reduced paging costs. Specifying both **copy_key** and **point_to_key** for the same index results in a **point_to_key** index.
- **os_index_path::point_to_key**: indicates an index with entries consisting of pointer-to-key-value / element pairs, as opposed to key-value / element pairs. For a **point_to_key** index, an entry includes a pointer to the object at the end of the **os_index_path** that specifies the key. Because of increased paging costs, such an index generally provides slower access times than an index that copies its keys; but a **point_to_key** index takes up less space. Specifying both **copy_key** and **point_to_key** for the same index results in a **point_to_key** index.
- **os_index_path::use_references**: indicates a reference-based (as opposed to pointer-based) index. For very large collections, using an **os_ixonly** representation and a reference-based index (or indexes) can, for many operations, significantly reduce address space consumption. Collections using *any* reference-based index must use *only* reference-based indexes.

By default, indexes have the following behavior:

os_index_path::unordered |
os_index_path::allow_duplicates |
os_index_path::copy_key.

The **os_segment*** argument points to the segment in which the new index is to be allocated. If the argument is omitted or if 0 is supplied, the index is allocated in the same segment as the collection being indexed. Putting each index in its own dedicated segment often results in better performance.

The function **add_index()** can be invoked at any point in the lifetime of a collection.

In a given database, the first time an unordered index is created for a particular key type, ObjectStore modifies the database's schema. Similarly, the first time an ordered index is created for a particular key type, ObjectStore modifies the schema. Schema modification write-locks segment **0**, which effectively locks the entire database.

```
void add_index(
    const os_index_path&,
    os_int32 options,
    os_database*
);
```

Creates an index into the specified collection, keyed by the specified **os_index_path**. The presence of the index allows optimization of queries involving lookup of collection elements based on the specified path. See the class **os_index_path** on page 179. If the specified collection already has an index with the specified key, this call is ignored. The **os_database*** argument points to the database in which the new index is to be allocated. See above for an explanation of the options argument.

```
void add_index(
    const os_index_path&,
    os_segment* = 0
);
```

Creates an index into the specified collection, keyed by the specified **os_index_path**. The presence of the index allows optimization of queries involving lookup of collection elements based on the specified path. See the class **os_index_path** on page 179. If the specified collection already has an index with the specified key, this call is ignored. The **os_segment*** argument points to the segment in which the new index is to be allocated. If the argument is omitted or if 0 is supplied, the index is allocated in the same segment as the collection being indexed.

```
void add_index(
    const os_index_path&,
    os_database*
);
```

Creates an index into the specified collection, keyed by the specified **os_index_path**. The presence of the index allows optimization of queries involving lookup of collection elements based on the specified path. See the class **os_index_path** on page 179. If the specified collection already has an index with the specified key, this call is ignored. The **os_database*** argument points to the database in which the new index is to be allocated.

os_collection::allow_duplicates

Possible disjunct of the bit-wise disjunction composing the **behavior** argument to the **create()** and **change_behavior()** members of **os_collection**, **os_Collection**, and their subtypes.

Indicates that the new or changed collection should allow duplicate elements, that is, multiple occurrences of the same element. Inserting a value into a collection that allows duplicates always increases the collection’s cardinality, and increases the count of that value in the collection. If duplicates are not allowed, insertion of an element that is already present either is silently ignored or signals the exception err_coll_duplicates. See **os_collection::insert()** on page 107. Allowing duplicates generally increases the efficiency of insert operations, since the operations do not have to check for the presence of the inserted element, as they do if duplicates are not allowed.

os_collection::allow_nulls

Possible disjunct of the bit-wise disjunction composing the **behavior** argument to the **create()** and **change_behavior()** members of **os_collection**, **os_Collection**, and their subtypes. Indicates that the new or changed collection should allow null elements, that is, 0. Inserting a value into a collection that disallows nulls signals the exception err_coll_nulls. See **os_collection::insert()** on page 107.

os_collection::associate_policy

Possible argument to **create()** and **change_rep()** members of **os_collection**, **os_Collection**, and their subtypes. Indicates that the **rep_policy** argument to **create()** or **change_rep()** should be used to determine how the new or changed collection’s representation changes in response to changes in cardinality. See also **os_collection::dont_associate_policy** on page 101.

os_collection::be_an_array

Possible disjunct of the bit-wise disjunction composing the **behavior** argument to the **create()** and **change_behavior()** members of **os_collection**, **os_Collection**, **os_list**, and **os_List**. For collections that maintain order only. With this behavior, access to the n^{th} element is an **O(1)** operation.

os_collection::cardinality()

os_unsigned_int32 cardinality() const;

Returns the sum of the count 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. See **os_ixonly_bc**.

os_collection::cardinality_is_maintained()

os_int32 cardinality_is_maintained() const;

Returns nonzero if the collection maintains cardinality; returns **0** otherwise. See **os_ixonly_bc**.

os_collection::change_behavior()

```
void change_behavior(
    os_unsigned_int32 behavior,
    os_int32 = verify
);
```

Changes the behavior of the specified collection.

The **behavior** is a bit pattern, the bit-wise disjunction (using the operator |) of enumerators indicating all the desired properties for the changed collection. The enumerators are

- **os_collection::allow_nulls**
- **os_collection::allow_duplicates**
- **os_collection::signal_duplicates**

- **os_collection::maintain_order**
- **os_collection::maintain_cursors**
- **os_collection::signal_cardinality**
- **os_collection::pick_from_empty_returns_null**

A run-time error is signaled if an attempt is made to change a collection to both signal and allow duplicates. A run-time error is signaled if an attempt is made to change an **os_bag** or **os_Bag** to disallow duplicates or be ordered, or to change an **os_set** or **os_Set** to allow duplicates or be ordered, or to change an **os_list** or **os_List** to be unordered.

os_collection::verify

When you change a collection so that it no longer allows duplicate or null insertions, you might want to check to see if duplicates or nulls are already present. Such a check is performed for you if you supply the enumerator **os_collection::verify** as the second argument. If nulls are found, err_coll_nulls is signaled. If duplicates are found, and **signal_duplicates** is on, err_coll_duplicates is signaled. If **signal_duplicates** is not on, the first among each set of duplicates is retained and trailing duplicates are silently removed. If **os_collection::verify** is not used, the resulting collection is assumed to be free of duplicates or nulls.

os_collection::change_rep()

```
void change_rep(
    os_unsigned_int32 expected_size,
    const os_coll_rep_descriptor *rep_policy = 0,
    os_int32 retain = dont_associate_policy
);
```

Changes the representation or representation policy of the specified collection.

The **expected_size** is the cardinality you expect the collection to have when fully loaded. This value is used by ObjectStore to determine the collection's initial representation. This saves on the overhead of transforming the collection's representation as it grows during loading.

The **rep_policy** is the representation policy to be associated with the collection until explicitly changed, if **retain** is **os_collection::associate_policy**. If **retain** is **os_collection::dont_associate_policy**, the **rep_policy** is used, together with the

expected_size, only to determine the collection's initial representation. (A representation policy is, essentially, a mapping from cardinality ranges to representation types — see **os_coll_rep_descriptor** on page 143, and in *ObjectStore Advanced C++ API User Guide* see **os_ptr_bag** and **os_packed_list**.)

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 zero) of the specified **void*** in the collection for which the function was called.

os_collection::create()

```
static os_collection &create(
    os_database *db,
    os_unsigned_int32 behavior = 0,
    os_int32 expected_size = 0,
    const os_coll_rep_descriptor *rep_policy = 0,
    os_int32 retain = dont_associate_policy
);
```

Creates a new collection in the database pointed to by **db**. If the transient database is specified, the collection is allocated in transient memory.

Properties

Every instance of **os_Collection** has the following properties:

- Its entries have no intrinsic order.
- Duplicate elements are not allowed.

By default a new **os_Collection** object also has the following properties:

- Performing **pick()** on an empty result of querying the collection raises err_coll_empty.

- 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.

Using the **behavior** argument, you can customize the behavior of new **os_Collection**s with regard to these last three properties. See Customizing Collection Representation in *ObjectStore Advanced C++ API User Guide*.

By default, instances of **os_Collection** are presized with a representation suitable for cardinality 20 or less. If you want a new collection presized for a different cardinality, supply the **expected_size** argument explicitly.

If you want to customize the representation of a new collection, see Customizing Collection Representation in *ObjectStore Advanced C++ API User Guide*.

The default representation policy for **os_Collection**s is as follows:

- As the collection grows from 0 to 15, the representation is **os_chained_list**.
- Once the collection grows past 15, **os_dyn_hash** is used.

Note that **expected_size** determines the initial representation. So, for example, if **expected_size** is 21, **os_dyn_hash** is used for the collection's entire lifetime (unless you use **change_rep()**.

```
static os_collection &create(
    os_segment * seg,
    os_unsigned_int32 behavior = 0,
    os_int32 expected_size = 0,
    const os_coll_rep_descriptor *rep_policy = 0,
    os_int32 retain = dont_associate_policy
);
```

Creates a new collection in the segment pointed to by **seg**. If the transient segment is specified, the collection is allocated in transient memory. The rest of the arguments are just as described above.

```
static os_collection &create(
    os_object_cluster *clust,
    os_unsigned_int32 behavior = 0,
    os_int32 expected_size = 0,
    const os_coll_rep_descriptor *rep_policy = 0,
```

```
    os_int32 retain = dont_associate_policy
);
```

Creates a new collection in the object cluster pointed to by **clust**. The rest of the arguments are just as described above.

```
static os_collection &create(
    void * proximity,
    os_unsigned_int32 behavior = 0,
    os_int32 expected_size = 0,
    const os_coll_rep_descriptor *rep_policy = 0,
    os_int32 retain = dont_associate_policy
);
```

Creates a new collection in the segment occupied by the object pointed to by **proximity**. If the object is part of an object cluster, the new collection is allocated in that cluster. If the specified object is transient, the collection is allocated in transient memory. The rest of the arguments are just as described above.

os_collection::default_behavior()

```
static os_unsigned_int32 default_behavior();
```

Returns a bit pattern indicating this type's default behavior.

os_collection::destroy()

```
static void destroy(os_collection&);
```

Deletes the specified collection and deallocates associated storage.

os_collection::dont_associate_policy

Possible argument to **create()** and **change_rep()** members of **os_collection**, **os_Collection**, and their subtypes. Indicates that the **rep_policy** argument to **create()** or **change_rep()** should be used, together with **expected_size**, only to determine the new or changed collection's initial representation. See also **os_collection::associate_policy** on page 96.

os_collection::dont_verify

Possible argument to **os_collection::change_behavior()**, when changing a collection to allow duplicates or nulls. If this enumerator is supplied, the changed collection is assumed to be free of duplicates and nulls. See also **os_collection::verify** on page 98.

os_collection::drop_index()

void drop_index(const os_index_path &p);

Destroys the index into the specified collection whose key is specified by **p**. The argument **p** does *not* need to be the same instance of **os_index_path** supplied when the index was added, but it must specify the same key. Two **os_index_path**s created with the same path string and type string specify the same index key.

Collections with large cardinality might warrant removing the index with multiple transactions. See **os_collection::multi_trans_ drop_index()** on page 111.

An err_no_such_index exception is signaled if an index with the specified key was never added to the collection.

os_collection::EQ

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

os_collection::empty()

os_int32 empty();

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

os_collection::exists()

```
os_int32 exists(
    char *element_type_name,
    char *query_string,
    os_database *schema_database = 0,
    char *filename,
    os_unsigned_int32
) const;
```

Returns a nonzero **os_int32** (true) if there exists an element of **this** that satisfies the selection criterion expressed by the **query_string**. Otherwise, **0** (false) is returned.

The argument **element_type_name** is the name of the element type of **this**. Names established through the use of **typedef** are not allowed.

The **query_string** is a C++ *control expression* indicating the query's selection criterion. An element **e** satisfies the selection criterion if the control expression evaluates to a nonzero **os_int32** (true) when **e** is bound to **this**.

Any string consisting of an **os_int32**-valued C++ expression is allowed, as long as

- There are no variables that are not data members.
- There are no function calls, except calls to **strcmp()** or **strcoll()**, calls involving a comparison operator for which the user has defined a corresponding rank function, calls to rank functions themselves, and calls to member functions that satisfy the restrictions described below.

Within the selection criterion of query expressions, member names are implicitly qualified by **this**, just as are member names in function member bodies.

Restrictions

Member functions called in query strings are subject to certain restrictions:

- The return type must be a pointer type or an arithmetic type (**int**, **char**, **float**, and so on — see *The Annotated C++ Reference Manual*, Section 3.6.1). If it is not, a compile-time error results.
- The function must take no arguments except the **this** argument. Otherwise, a compile-time error results.

To perform a query, ObjectStore sometimes (depending on what indexes are present) issues calls to member functions used in paths and queries. If such a member function allocates memory it does not free (for example if it returns a pointer to newly allocated memory), memory leaks can result; ObjectStore does not free the space the function allocates. So member functions used in paths or queries should not allocate persistent memory or memory in the transient heap.

Member function in a query string

Applications that use a member function (*not* returning a reference) in a query string must do four things:

- Define an **os_backptr**-valued data member in the class that defines the member function.
- Call the macro **os_query_function()**. This should be defined at file scope, for example, in the header file that contains the class

that defines the member function. See **os_query_function()** for more information.

- Call the macro **os_query_function_body()**. This should be defined at file scope in a source file that will only be compiled into the application once. See **os_query_function_body()** for more information.

Member function, returning a reference, in a query string

For applications that use a member function that returns a reference in a query string, you must do the following four things:

- Define an **os_backptr**-valued data member in the class that defines the member function.
- Call the macro **os_query_function_returning_ref()**. This should be defined at file scope, for example, in the header file that contains the class that defines the member function. See **os_query_function_body_returning_ref()** for more information.
- Call the macro **os_query_function_body_returning_ref()**. This should be defined at file scope in a source file that will only be compiled into the application once. See **os_query_function_body_returning_ref()** for more information.

To maintain indexes keyed by paths containing member function calls, use **os_backptr::make_link()** and **os_backptr::break_link()**.

Nested queries

The query string can itself contain queries. A notation is defined to allow the user to specify such nested queries conveniently in a single call to a query member function.

A nested collection-valued query has the form

collection-expression **[:** *os_int32-expression* **:]**

where *collection-expression* is an expression of type **os_collection**, and *os_int32-expression* is the selection criterion for the nested query.

A nested single-element query has the form

collection-expression **[%** *os_int32-expression* **%]**

where *collection-expression* and *os_int32-expression* are as for nested collection-valued queries. This form evaluates to one element of *collection-expression*. If there is more than one element that satisfies the nested query's selection criterion, one of them is picked and returned. If no element satisfies the query, an err_coll_empty exception is signaled.

The **schema_database** is a database whose schema contains all the types mentioned in the selection criterion. This database provides the environment in which the query is analyzed and optimized. The database in which the collection resides is often appropriate.

If the transient database is specified, the application's schema (stored in the application schema database) is used to evaluate the query.

os_int32 exists(const os_bound_query&) const;

Returns a nonzero **os_int32** (true) if there exists an element of **this** that satisfies the **os_bound_query**. Otherwise, **0** (false) is returned.

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.

os_collection::get_indexes()

os_collection *get_indexes() const;

If **this** has associated indexes, returns a collection of **os_index_name***s, one for each index. If **this** has no indexes, **0** is returned. The caller is responsible for deleting the collection and its contents.

os_collection::get_rep()

const os_coll_rep_descriptor &get_rep() const;

Returns a pointer to the specified collection's currently active rep descriptor.

os_collection::get_thread_locking()

static os_boolean get_thread_locking();

If nonzero is returned, collections thread locking is enabled; if **0** is returned, collections thread locking is disabled. See **os_collection::set_thread_locking()** on page 127.

os_collection::has_index()

```
os_int32 has_index(
    const os_index_path&,
    os_int32 index_options = unordered
) const;
```

Returns a value saying whether an index can support the index type specified with **index_options**. Possible values for **index_option** are **ordered** and **unordered**.

You must supply a path string and one of the index options. An index that supports exact match queries (hash table) can only be used for exact matches. An index that supports range queries (binary tree) can be used for both exact match and range queries. In effect, **os_collection::has_index** answers the question "can this index support this type of query" and not what option was used to create the index.

- For an index created with the ordered option, the following is true:

has_index(path,os_index::ordered)	Returns true
has_index(path,os_index::unordered)	Returns true

- For an index created with the unordered option, the following is true:

has_index(path,os_index::ordered)	Returns false
has_index(path,os_index::unordered)	Returns true

Returns a nonzero **os_int32** (true) if the collection has an index that supports the functionality of an index with the given options. That is, an ordered index (one that supports queries of the form "all things greater than X") can be used as an unordered index (one that only supports queries of the form "all things equal to X"). However, an unordered index cannot be used as an ordered index. See **os_collection::add_index()** on page 92 for additional information.

os_collection::initialize()

static void initialize();

Must be executed in a process before any use of ObjectStore collection or relationship functionality is made. After the first execution of **initialize()** in a given process, subsequent executions in that process have no effect.

os_collection::insert()

void insert(const void*);

Adds the specified **void*** to the collection for which the function was called. If the collection is ordered, the element is inserted at the end of the collection. If the collection disallows and signals duplicates, and the specified **void*** is already present in the collection, *err_coll_duplicates* is signaled. If the collection disallows duplicates and does not signal 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_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. 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, err_coll_nulls 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. 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, err_coll_nulls is signaled.

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_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. 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, err_coll_nulls 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. 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, err_coll_nulls is signaled.

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. 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, err_coll_nulls is signaled.

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. 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, err_coll_nulls is signaled.

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_cursors

Possible element of the bit-wise disjunction that makes up the **behavior** argument to the **create()** and **change_behavior()** members of **os_collection**, **os_Collection**, and their subtypes. Indicates that the new or changed collection should support updates during iteration. If specified for an unordered collection, iterations over the collection that use safe cursors are guaranteed to visit elements that are inserted during the iteration. If specified for an ordered collection, iterations over the collection that use safe cursors are guaranteed to visit elements that are inserted during the iteration, provided the element was inserted later in the cursor’s associated order than the cursor’s position at the time of the insertion. See also **os_cursor::safe** on page 149.

os_collection::maintain_order

Possible element of the bit-wise disjunction that makes up the **behavior** argument to the **create()** and **change_behavior()** members of **os_collection**, **os_Collection**, and their subtypes. Indicates that the new or changed collection should maintain its elements in their order of insertion with **insert()**. 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::multi_trans_add_index()

```
static void multi_trans_add_index(
    os_reference c,
    const os_index_path & p,
    os_int32 index_options,
    os_segment * index_seg,
    os_segment * scratch_seg,
    os_unsigned_int32 num_per_trans );
```

Creates an index into the collection specified by the **os_reference c**, keyed by the specified **os_index_path**. This function adds the given index to the given collection using multiple transactions. Until the index is fully added, it is unusable. That is, the index raises an exception if an attempt is made to insert or remove through other means. This implies that the collection is effectively write-locked until all the transactions needed to add the index commit.

Function arguments

- **c** is an **os_reference** to the collection to which to add the index.
- **p** is an **os_index_path**.
- **index_options** is the same as it is for **add_index**.
- **index_seg** is the segment in which to create the index (just like **add_index**).
- **scratch_seg** is a segment that is used internally and can be deleted when the function returns. It cannot be **os_segment::get_transient_segment()**.
- **num_per_trans** is the number of collection elements to insert into the collection per transaction.

If the **multi_trans_add_index** operation fails partway through, **os_collection::drop_index()** can be used.

os_collection::multi_trans_drop_index()

static void multi_trans_drop_index(
os_reference c,
const os_index_path & p,
os_segment * scratch_seg,
os_unsigned_int32 num_per_trans);

Destroys the index into the specified collection whose key is specified by **p**. This differs from **os_collection::drop_index()** only in that index maintenance is done using multiple transactions.

The index is unusable while it is being removed. The index raises an exception if an attempt is made to insert or remove it by other means. This means that the collection is effectively write-locked until all the transactions needed to remove the index commit.

Function arguments

- **c** is an **os_reference** to the collection from which the index should be removed.
- **p** is an **os_index_path**.
- **scratch_seg** is the segment that is used internally and can be deleted when the function returns. It cannot be **os_segment::get_transient_segment()**.
- **num_per_trans** is the number of collection elements to update per transaction.

Ifs the **multi_trans_drop_index** operation fails partway through, **os_collection::drop_index()** can be used.

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, err_coll_empty is signaled, unless the collection has behavior **os_collection::pick_from_empty_returns_null**, in which case **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**.

Note: The assignment operator semantics are described below in terms of insert operations into the target collection. Describing the semantics in terms of insert operations serves to illustrate how duplicate, null, and order semantics are enforced. The actual implementation of the assignment might be quite 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. The result does not maintain cursors or signal duplicates.

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. The result does not maintain cursors or signal duplicates.

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. The result does not maintain cursors or signal duplicates.

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. The result does not maintain cursors or signal duplicates.

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. The result does not maintain cursors or signal duplicates.

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. The result does not maintain cursors or signal duplicates.

os_collection::order_by_address

Possible argument to 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_collection::ordered

Used as an argument to **os_collection::add_index**, to specify that an ordered index (B-tree) is to be maintained. Use of **os_index_path::ordered** is now preferred.

os_collection::pick()

void* pick() const;

Returns an arbitrary element of the specified collection. If the collection is empty, err_coll_empty is signaled, unless the collection has behavior **os_collection::pick_from_empty_returns_null**, in which case **0** is returned.

```
void* pick(
    const os_index_path &path,
    const os_coll_range &range
) const;
```

Returns an element of the specified collection such that the result of applying **path** to the element is a value that satisfies **range** (see the class **os_coll_range** on page 138). If there is no such element, err_coll_empty is signaled, unless the collection has behavior **pick_from_empty_returns_null**, in which case **0** is returned.

os_collection::pick_from_empty_returns_null

Possible disjunct of the bit-wise disjunction composing the **behavior** argument to the **create()** and **change_behavior()** members of **os_collection**, **os_Collection**, and their subtypes. Indicates that **only()** and **pick()** should return **0** when performed on empty collections. Without this behavior, performing these on empty collections provokes the exception err_coll_empty.

os_collection::query()

```
os_collection &query(
    char *element_type_name,
    char *query_string,
    os_database *schema_database = 0,
    char *file_name = 0,
    os_unsigned_int32 line = 0,
    os_boolean dups = query_dont_preserve_duplicates
) const;
```

Returns a reference to a heap-allocated collection containing those elements of **this** that satisfy the selection criterion expressed by the **query_string**. When you no longer need the resulting collection, you should reclaim its memory with **::operator delete()** to avoid memory leaks.

The argument **element_type_name** is the name of the element type of **this**. Names established through the use of **typedef** are not allowed.

The **query_string** is a C++ *control expression* indicating the query's selection criterion. An element, **e**, satisfies the selection criterion if the control expression evaluates to a nonzero **os_int32** (true) when **e** is bound to **this**.

Any string consisting of an **os_int32**-valued C++ expression is allowed, as long as

- There are no variables that are not data members.

- There are no function calls, except calls to **strcmp()** or **strcoll()**, calls involving a comparison operator for which the user has defined a corresponding rank function, and calls to member functions that satisfy the restrictions listed below.

Within the selection criterion of query expressions, member names are implicitly qualified by **this**, just as are member names in function member bodies.

Restrictions

Member functions called in query strings are subject to certain restrictions:

- The return type must be a pointer type or an arithmetic type (**int**, **char**, **float**, and so on — see *The Annotated C++ Reference Manual*, Section 3.6.1). If it is not, a compile-time error results.
- The function must take no arguments except the **this** argument. Otherwise, a compile-time error results.

To perform a query, ObjectStore sometimes (depending on what indexes are present) issues calls to member functions used in paths and queries. If such a member function allocates memory it does not free (for example, if it returns a pointer to newly allocated memory), memory leaks can result; ObjectStore does not free the space the function allocates. So member functions used in paths or queries should not allocate persistent memory or memory in the transient heap.

Member function in a query string

Applications that use a member function (*not* returning a reference) in a query string must do four things:

- Define an **os_backptr**-valued data member in the class that defines the member function.
- Call the macro **os_query_function()**. This should be defined at file scope, for example, in the header file that contains the class that defines the member function. See **os_query_function()** for more information.
- Call the macro **os_query_function_body()**. This should be defined at file scope in a source file that will only be compiled into the application once. See **os_query_function_body()** for more information.
- Call the macro **OS_MARK_QUERY_FUNCTION()**. This macro should be invoked in the schema source file. See **OS_MARK_ QUERY_FUNCTION()** on page 268 for more information.

Member function, returning a reference, in a query string

For applications that use a member function that returns a reference in a query string, you must do the following four things:

- Define an **os_backptr-**valued data member in the class that defines the member function.
- Call the macro **os_query_function_returning_ref()**. This should be defined at file scope, for example, in the header file that contains the class that defines the member function. See **os_query_function_body_returning_ref()** for more information.
- Call the macro **os_query_function_body_returning_ref()**. This should be defined at file scope in a source file that will only be compiled into the application once. See **os_query_function_body_returning_ref()** for more information.
- Call the macro **OS_MARK_QUERY_FUNCTION()**. This macro should be invoked in the schema source file. See **OS_MARK_QUERY_FUNCTION()** on page 268 for more information.

Within the selection criterion of query expressions, member names are implicitly qualified by **this**, just as are member names in function member bodies.

Nested queries

The query string can itself contain queries. A notation is defined to allow the user to specify such nested queries conveniently in a single call to a query member function.

A nested collection-valued query has the form

collection-expression **[:** *os_int32-expression* **:]**

where *collection-expression* is an expression of type **os_collection**, and *os_int32-expression* is the selection criterion for the nested query.

A nested single-element query has the form

collection-expression **[%** *os_int32-expression* **%]**

where *collection-expression* and *os_int32-expression* are as for nested collection-valued queries. This form evaluates to one element of *collection-expression*. If there is more than one element that satisfies the nested query's selection criterion, one of them is picked and returned.

A nested query returning a collection is converted to an **os_int32** when appropriate, using **os_collection::operator os_int32()**.

The **schema_database** is a database whose schema contains all the types mentioned in the selection criterion. This database provides the environment in which the query is analyzed and optimized. The database in which the collection resides is often appropriate.

ObjectStore uses **file_name** and **line** when reporting errors related to the query. You can set them to identify the location of the query's source code.

If **dups** is the enumerator **query_dont_preserve_duplicates**, duplicate elements that satisfy the query condition are not included in the query result. If **dups** is the enumerator **query_preserve_duplicates**, duplicate elements that satisfy the query condition are included in the query result.

If the transient database is specified, the application's schema (stored in the application schema database) is used to evaluate the query.

```
os_collection &query(
    const os_bound_query&,
    os_boolean dups = query_dont_preserve_duplicates
) const;
```

Returns a reference to a heap-allocated collection containing those elements of **this** that satisfy the **os_bound_query**. If **dups** is the enumerator **query_dont_preserve_duplicates**, duplicate elements that satisfy the query condition are not included in the query result. If **dups** is the enumerator **query_preserve_duplicates**, duplicate elements that satisfy the query condition are included in the query result.

When you no longer need the resulting collection, you should reclaim its memory with **::operator delete()** to avoid memory leaks.

os_collection::query_pick()

```
void *query_pick(
    char *element_type_name,
    char *query_string,
    os_database *schema_database = 0,
    char *filename,
    os_unsigned_int32 line=0
) const;
```

Returns an element of **this** that satisfies the selection criterion expressed by the **query_string**. If there is more than one such element, one is picked arbitrarily and returned. If no element satisfies the query or the collection is empty, **0** is returned.

The argument **element_type_name** is the name of the element type of **this**. Names established through the use of **typedef** are not allowed.

The **query_string** is a C++ *control expression* indicating the query's selection criterion. An element, **e**, satisfies the selection criterion if the control expression evaluates to a nonzero **os_int32** (true) when **e** is bound to **this**.

Any string consisting of an **os_int32-**valued C++ expression is allowed, as long as

- There are no variables that are not data members.
- There are no function calls, except calls to **strcmp()** or **strcoll()**, calls involving a comparison operator for which the user has defined a corresponding rank function, and calls to member functions that satisfy the restrictions listed below.

Within the selection criterion of query expressions, member names are implicitly qualified by **this**, just as are member names in function member bodies.

Member functions called in query strings are subject to certain restrictions:

- The return type must be a pointer type or an arithmetic type (**int**, **char**, **float**, and so on — see *The Annotated C++ Reference Manual,* Section 3.6.1). If it is not, a compile-time error results.
- The function must take no arguments except the **this** argument. Otherwise, a compile-time error results.

To perform a query, ObjectStore sometimes (depending on what indexes are present) issues calls to member functions used in paths and queries. If such a member function allocates memory it does not free (for example, if it returns a pointer to newly allocated memory), memory leaks can result; ObjectStore does not free the space the function allocates. So member functions used in paths or queries should not allocate persistent memory or memory in the transient heap.

Member function in a query string

Applications that use a member function (*not* returning a reference) in a query string must do four things:

- Define an **os_backptr-**valued data member in the class that defines the member function.
- Call the macro **os_query_function()**.
- Call the macro **os_query_function_body()**.
- Call the macro **OS_MARK_QUERY_FUNCTION()**.

Member function, returning a reference, in a query string

For applications that use a member function that returns a reference in a query string, you must do the following four things:

- Define an **os_backptr-**valued data member in the class that defines the member function.
- Call the macro **os_query_function_returning_ref()**.
- Call the macro **os_query_function_body_returning_ref()**.
- Call the macro **OS_MARK_QUERY_FUNCTION()**.

Within the selection criterion of query expressions, member names are implicitly qualified by **this**, just as are member names in function member bodies.

Nested queries

The query string can itself contain queries. A notation is defined to allow the user to specify such nested queries conveniently in a single call to a query member function.

A nested collection-valued query has the form

collection-expression **[:** *os_int32-expression* **:]**

where *collection-expression* is an expression of type **os_collection**, and *os_int32-expression* is the selection criterion for the nested query.

A nested single-element query has the form

collection-expression **[%** *os_int32-expression* **%]**

where *collection-expression* and *os_int32-expression* are as for nested collection-valued queries. This form evaluates to one element of *collection-expression*. If there is more than one element that satisfies the nested query's selection criterion, one is picked arbitrarily and returned. If no element satisfies the query, **0** is returned.

The **schema_database** is a database whose schema contains all the types mentioned in the selection criterion. This database provides the environment in which the query is analyzed and optimized. The database in which the collection resides is often appropriate.

If the transient database is specified, the application's schema (stored in the application schema database) is used to evaluate the query.

void *query_pick(const os_bound_query&) const;

Returns an element of **this** that satisfies the **os_bound_query**. If there is more than one such element, one is picked arbitrarily and returned. If no element satisfies the query or the collection is empty, **0** is returned.

os_collection::remove()

os_int32 remove(const void*);

Removes the specified **void*** from the collection for which the function was called, if the **void*** is an element of the collection. If the collection is ordered, the first occurrence of the specified **void*** is removed. Returns a nonzero **os_int32** if an element was removed, **0** otherwise.

os_collection::remove_at()

void remove_at(const os_cursor&);

Removes from the specified collection the element at which the cursor is positioned. The position of all elements after the removed element decreases 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 not positioned at an element, err_coll_illegal_cursor is signaled. If the collection is not ordered, err_coll_not_supported is signaled.

void remove_at(os_unsigned_int32 position);

Removes from the specified collection the element with the specified position. The position of all elements after the removed element decreases by 1. 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::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, **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.

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_first()**, 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.

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.

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 other overloading of this function, below. 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 other overloading of this function, below. 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::set_query_memory_mode()

static void set_query_memory_mode(
os_query_memory_mode mode);

os_query_memory_mode is an enumeration type whose enumerators are:

os_query_memory_mode_none — Use this mode when you know you are doing small or well optimized queries and do not want to incur even a small amount of overhead. This mode is the default when the query is being executed in a nested transaction.

os_query_memory_mode_normal — Marks the address space at the start of the query. If at any time during the query address space runs out, the query is restarted from the beginning using low memory mode.

Use this mode if your application typically does not run queries that use large amounts of address space, but you still want to safeguard against running out of address space. This mode is the default for queries being executed in nonnested transactions.

os_query_memory_mode_low — Marks the address space at the start of the query. Put the results in a collection of references. Catch the err_address_space_full exception inside the query processor. When the exception is signaled, release the address space and continue the query from where it left off. At the end of the query, a final release is performed.

Use this mode for running large queries that can use a great deal of address space. This mode requires some overhead, but ensures the query completes without restarting.

os_collection::set_thread_locking()

static void set_thread_locking(os_boolean);

Collections thread locking is enabled by default when you link with a threads library. To enable collections thread locking explicitly, pass a nonzero value. *ObjectStore thread locking must be* *enabled at the time of the call for this to have any effect*. To disable collections thread locking, pass 0 to this function.

If your application uses multiple threads, and the synchronization coded in your application allows two threads to be within the collections or queries libraries at the same time, you need collections thread locking enabled. See also **objectstore::set_thread_locking()**.

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. Before you add a new index to an **os_ixonly_bc** collection, call this function. If you do not, **add_index()** will work correctly, but less efficiently than if you do.

os_collection_size

This class serves as a formal argument to an overloading of the collection subtype constructors. It is used to specify the new collection's expected size. An integer type is not used as the formal in order to prevent certain undesirable conversions and conversion ambiguities. The actual argument supplied to the collection constructors can be an **os_int32**, since **os_collection_size** defines a conversion constructor, **os_collection_size::os_collection_size(os_int32)**.

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.

os_coll_query

Instances of this class are query objects. For more information on query objects, see Preanalyzed Queries in the *ObjectStore Advanced C++ API User Guide*

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.

os_coll_query::create()

```
static const os_coll_query &create(
    const char *element_type,
    const char *query_string,
    os_database *schema_database,
    os_boolean cache_query = 0,
    char *file_name = 0,
    os_unsigned_int32 line = 0
);
```

Creates a query object, possibly with free variables and function references. The query object can be used to create an **os_bound_query**, which can then be executed with **os_collection::query()**.

The argument **element_type** is the name of the element type of **this**. Names established through the use of **typedef** are not allowed.

The **query_string** is a C++ *control expression* indicating the query's selection criterion. An element, **e**, satisfies the selection criterion if the control expression evaluates to a nonzero **os_int32** (true) when **e** is bound to **this**. Only some kinds of function calls are allowed in the query string. See "Restrictions on member functions in query strings" on page 131 for more information.

The **schema_database** is a database whose schema contains all the types mentioned in the selection criterion. This database provides the environment in which the query is analyzed and optimized. The database in which the collection resides is often appropriate. If the transient database is specified, the application's schema (stored in the application schema database) is used to evaluate the query.

If **cache_query** is a nonzero **os_int32** (true), the query object is allocated in the schema segment of the database specified. If the database specified is the transient database, the object is allocated in the schema segment of the application schema database. If **cache_query** is zero (the default), the object is transiently allocated and the user is responsible for deleting it.

file_name, if supplied, should be the name of the source file containing the call to **create()**. It is used only if an error is signaled during query analysis. Its sole purpose is to allow the resulting error message to make reference to the source file containing the code that caused an error.

line, if supplied, should be the number of the line in the source file on which the call to **create()** appears. It is used only if an error is signaled during query analysis. Its sole purpose is to allow the resulting error message to make reference to the source file line containing the code that caused an error.

Restrictions on member functions in query strings

Any string consisting of an **os_int32**-valued C++ expression is allowed in a query string, as long as

- Variables are also data members of the elements of the collection.
- For local variables (free references), you create an **os_coll_query** object.
- For global functions (free references), you create an **os_coll_query** object.
- There are no function calls, except calls to **strcmp()** or **strcoll()**,
- There are no comparison operators for which the user might be required to define a corresponding rank / hash function.
- There are no calls to member functions that satisfy the restrictions listed below.

Within the selection criterion of query expressions, member names are implicitly qualified by **this**, just as are member names in function member bodies.

Restrictions

Functions called in query strings are subject to certain restrictions:

- The return type can be a basic type (**int**, **char**, **float**, **char***).
- If the function is a member function it can also return a pointer or a reference to a class type.

- The function can take up to two arguments. The first argument must be a pointer. For member functions **this** is the implied first argument.
- Global functions are free references and must be used in an **os_coll_query** object.
- Member functions can be used like data members.

To perform a query, ObjectStore sometimes (depending on what indexes are present) issues calls to member functions used in paths and queries. If such a member function allocates memory it does not free (for example if it returns a pointer to newly allocated memory), memory leaks can result; ObjectStore does not free the space the function allocates. So member functions used in paths or queries should not allocate persistent memory or memory in the transient heap.

Member function in a query string

Applications that use a member function (*not* returning a reference) in a query string must do four things:

- Define an **os_backptr**-valued data member in the class that defines the member function. It must precede the member function declaration in the class definition.
- Call the macro **os_query_function()**. This should be defined at file scope, for example, in the header file that contains the class that defines the member function. See **os_query_function()** for more information.
- Call the macro **os_query_function_body()**. This should be defined at file scope in a source file that will only be compiled into the application once. See **os_query_function_body()** for more information.
- Call the macro **OS_MARK_QUERY_FUNCTION()**. This macro should be invoked in the schema source file. See **OS_MARK_QUERY_FUNCTION()** on page 268 for more information.

Member function, returning a reference, in a query string

For applications that use a member function that returns a reference in a query string, you must do the following four things:

- Define an **os_backptr**-valued data member in the class that defines the member function.
- Call the macro **os_query_function_returning_ref()**. This should be defined at file scope, for example, in the header file that

contains the class that defines the member function. See **os_query_function_body_returning_ref()** for more information.

- Call the macro **os_query_function_body_returning_ref()**. This should be defined at file scope in a source file that will only be compiled into the application once. See **os_query_function_body_returning_ref()** for more information.
- Call the macro **OS_MARK_QUERY_FUNCTION()**. This macro should be invoked in the schema source file. See **OS_MARK_QUERY_FUNCTION()** on page 268 for more information.

To maintain indexes keyed by paths containing member function calls, use **os_backptr::make_link()** and **os_backptr::break_link()**.

The query string can itself contain queries. A notation is defined to allow the user to conveniently specify such nested queries in a single call to a query member function.

A nested collection-valued query has the form

collection-expression **[:** *os_int32-expression* **:]**

where *collection-expression* is an expression of type **os_Collection**, and *os_int32-expression* is the selection criterion for the nested query.

A nested single-element query has the form

collection-expression **[%** *os_int32-expression* **%]**

where *collection-expression* and *os_int32-expression* are as for nested collection-valued queries. This form evaluates to one element of *collection-expression*. If there is more than one element that satisfies the nested query's selection criterion, one of them is picked and returned.

A nested query returning a collection is converted to an **os_int32** when appropriate, using **os_collection::operator os_int32()**.

```
static const os_coll_query &create(
    const char *element_type,
    const char *query_string,
    os_segment *schema,
    os_boolean cache_query = 0,
    char *file_name = 0,
    os_unsigned_int32 line = 0
);
```

Creates a query object, possibly with free variables and function references. The query object can be used to create an **os_bound_query**. The arguments are the same as those for the previous version of **create()**, except the **schema** database is specified with a pointer to one of its segments.

```
static const os_coll_query &create(
    const char *element_type,
    const char *query_string,
    void *schema,
    os_boolean cache_query = 0,
    char *file_name = 0,
    os_unsigned_int32 line = 0
);
```

Creates a query object, possibly with free variables and function references. The query object can be used to create an **os_bound_query**. The arguments are the same as those for the previous version of **create()**, except the **schema** database is specified with a pointer to an object it contains.

os_coll_query::create_exists()

```
static const os_coll_query &create_exists(
    const char *element_type,
    const char *query_string,
    os_database *schema_database,
    os_boolean cache_query = 0,
    char *file_name = 0,
    os_unsigned_int32 line = 0
);
```

Creates an existential query object, possibly with free variables and function references. The query object can be used to create an **os_bound_query**, which can then be executed with **os_collection::exists()**. The arguments are the same as those for **os_coll_query::create()** on page 130, except the schema database is specified with a pointer to its **os_database**.

```
static const os_coll_query &create_exists(
    const char *element_type,
    const char *query_string,
    os_segment *schema_database_segment,
    os_boolean cache_query = 0,
    char *file_name = 0,
    os_unsigned_int32 line = 0
);
```

Creates an existential query object, possibly with free variables and function references. The query object can be used to create an **os_bound_query**, which can then be executed with **os_collection::exists()**. The arguments are the same as those for the previous version of **create_exists()**, except the schema database is specified with a pointer to one of its segments.

```
static const os_coll_query &create_exists(
    const char *element_type,
    const char *query_string,
    void *schema_database_object,
    os_boolean cache_query = 0,
    char *file_name = 0,
    os_unsigned_int32 line = 0
);
```

Creates an existential query object, possibly with free variables and function references. The query object can be used to create an **os_bound_query**, which can then be executed with **os_collection::exists()**. The arguments are the same as those for the previous version of **create_exists()**, except the schema database is specified with a pointer to an object it contains.

os_coll_query::create_pick()

```
static const os_coll_query &create_pick(
    const char *element_type,
    const char *query_string,
    os_database *schema_database,
    os_boolean cache_query = 0,
    char *file_name = 0,
    os_unsigned_int32 line = 0
);
```

Creates a single-element query object, possibly with free variables and function references. The query object can be used to create an **os_bound_query**, which can then be executed with **os_collection::query_pick()**. The arguments are the same as those for **os_coll_query::create()**.

```
static const os_coll_query &create_pick(
    const char *element_type,
    const char *query_string,
    os_segment *schema_database_segment,
    os_boolean cache_query = 0,
    char *file_name = 0,
    os_unsigned_int32 line = 0
);
```

Creates a single-element query object, possibly with free variables and function references. The query object can be used to create an **os_bound_query**, which can then be executed with **os_ collection::query_pick()**. The arguments are the same as those for the previous version of **create_pick()**, except the schema database is specified with a pointer to one of its segments.

```
static const os_coll_query &create_pick(
    const char *element_type,
    const char *query_string,
    void *schema_database_object,
    os_boolean cache_query = 0,
    char *file_name = 0,
    os_unsigned_int32 line = 0
);
```

Creates a single-element query object, possibly with free variables and function references. The query object can be used to create an **os_bound_query**, which can then be executed with **os_ collection::query_pick()**. The arguments are the same as those for the previous version of **create_pick()**, except the schema database is specified with a pointer to an object it contains.

os_coll_query::destroy()

```
static void destroy(const os_coll_query&);
```

Deletes the specified instance of **os_coll_query**. This is the same as calling **delete** for the **os_coll_query** object.

os_coll_query::get_element_type()

```
const char *get_element_type() const;
```

Returns a string naming the element type supplied when the specified **os_coll_query** was created.

os_coll_query::get_query_string()

```
const char *get_query_string() const;
```

Returns the query string supplied when the specified **os_coll_ query** was created.

os_coll_query::get_file_name()

```
const char *get_file_name() const;
```

Returns the file name supplied when the specified **os_coll_query** was created.

os_coll_query::get_line_number()

os_unsigned_int32 get_line_number() const;

Returns the line number supplied when the specified **os_coll_query** was created.

os_coll_range

An instance of this class can be used to represent a selection criterion for collection elements. Each **os_coll_range** is associated with either a particular value or range of values. They can be used as argument to the **os_Cursor** constructor to create a restricted cursor, or as arguments to **os_Dictionary::pick()**.

os_coll_range::os_coll_range()

The constructor for **os_coll_range** has several overloadings. Each overloading falls into one of the following two groups:

- Overloadings that specify a lower bound only or an upper bound only (for example, "all values less than or equal to 7")
- Overloadings that specify both a lower and upper bound on a range of values (for example, "all values greater than 4 and less than or equal to 7")

In each of these two groups, there is one overloading for each C++ fundamental type of value, and one for the type **void***. To specify a range for any type of pointer value, use a **void*** overloading and pass a pointer to the value to serve as upper or lower bound.

Overloadings that specify a boundary only

```
os_coll_range(
    os_collection::restriction rel_op,
    unsigned char value
);
```

```
os_coll_range(
    os_collection::restriction rel_op,
    short value
);
```

```
os_coll_range(
    os_collection::restriction rel_op,
    os_signed_int8
);
```

```
os_coll_range(
    os_collection::restriction rel_op,
    unsigned short value
);
```

```
os_coll_range(
    os_collection::restriction rel_op,
    int value
);
```

```
os_coll_range(
    os_collection::restriction rel_op,
    unsigned int value
);

os_coll_range(
    os_collection::restriction rel_op,
    long value
);

os_coll_range(
    os_collection::restriction rel_op,
    float value
);

os_coll_range(
    os_collection::restriction rel_op,
    double value
);

os_coll_range(
    os_collection::restriction rel_op,
    long double value
);

os_coll_range(
    os_collection::restriction rel_op,
    const void* value
);
```

These construct an **os_coll_range** satisfied by all values that bear the relation **rel_op** to **value**. The argument **rel_op** should be coded as one of the following enumerators:

- **os_collection::EQ** (equal to)
- **os_collection::NE** (not equal to)
- **os_collection::LT** (less than)
- **os_collection::LE** (less than or equal to)
- **os_collection::GT** (greater than)
- **os_collection::GE** (greater than or equal to)

Overloadings that specify a range

```
os_coll_range(
    os_collection::restriction rel_op1,
    unsigned char value1,
    os_collection::restriction rel_op2,
    unsigned char value2);

os_coll_range(
    os_collection::restriction rel_op1,
    int value1,
    os_collection::restriction rel_op2,
```

```
    int value2);

os_coll_range(
    os_collection::restriction rel_op1,
    unsigned int value1,
    os_collection::restriction rel_op2,
    unsigned int value2
);

os_coll_range(
    os_collection::restriction rel_op1,
    short value1,
    os_collection::restriction rel_op2,
    short value2
);

os_coll_range(
    os_collection::restriction rel_op1,
    unsigned short value1,
    os_collection::restriction rel_op2,
    unsigned short value2
);

os_coll_range(
    os_collection::restriction rel_op1,
    os_signed_int8 value1,
    os_collection::restriction rel_op2,
    os_signed_int8 value2
);

os_coll_range(
    os_collection::restriction rel_op1, long value1,
    os_collection::restriction rel_op2, long value2
);

os_coll_range
    os_collection::restriction rel_op1,
    unsigned long value1,
    os_collection::restriction rel_op2,
    unsigned long value2
);

os_coll_range
    os_collection::restriction rel_op1,
    float value1,
    os_collection::restriction rel_op2,
    float value2
);

os_coll_range
    os_collection::restriction rel_op1,
    double value1,
    os_collection::restriction rel_op2,
    double value2
```

```
);

os_coll_range
    os_collection::restriction rel_op1,
    long double value1,
    os_collection::restriction rel_op2,
    long double value2
);

os_coll_range
    os_collection::restriction rel_op1,
    const void *value1,
    os_collection::restriction rel_op2,
    const void *value2
);
```

Each of these constructs an **os_coll_range** satisfied by all values that bear both the relation **rel_op1** to **value1** and the relation **rel_op2** to **value2**. The arguments **rel_op1** and **rel_op2** should be one of the following enumerators:

Enumerator	***Meaning***
os_collection::EQ	Equal to
os_collection::NE	Not equal to
os_collection::LT	Less than
os_collection::LE	Less than or equal to
os_collection::GT	Greater than
os_collection::GE	Greater than or equal to

Examples

When the value type is **char***, these relations are defined in terms of **strcmp()**. When the value type is a pointer to a user-defined class object, the user must supply rank / hash functions.

The following example is satisfied by all **int**s greater than 4 and less than or equal to 7.

```
os_coll_range( os_collection::GT, 4, os_collection::LE, 7 )
```

Do not specify the null range, for example,

```
os_coll_range( os_collection::LT, 4, os_collection::GT, 7 )
```

Do not specify a discontinuous range, for example,

```
os_coll_range( os_collection::GT, 4, os_collection::NE, 7 )
```

If you do, the exception err_am is signaled, and the following message is issued:

No handler for exception:
<maint-0023-0001>invalid restriction on unordered index (err_am)

os_coll_rep_descriptor

The class **os_coll_rep_descriptor** has no direct instances. Each instance is a direct instance of one of its subtypes:

os_chained_list_descriptor	**os_ptr_bag_descriptor**
os_ixonly_bc_descriptor	**os_ptr_hash_descriptor**
os_ixonly_descriptor	**os_tinyarray_descriptor**
os_packed_list_descriptor	**os_ordered_ptr_hash_descriptor**

Each instance has an associated cardinality range. In addition, each instance can contain a pointer to another **os_coll_rep_descriptor**. A list of descriptors linked together in this way designates a representation policy, a mapping from cardinality to representation type. How a collection's representation changes in response to cardinality changes is determined by the policy (if any) associated with that collection.

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.

os_coll_rep_descriptor::allowed_behavior()

os_unsigned_int32 allowed_behavior() const;

Returns a bit-wise disjunction of enumerators indicating this representation's allowed behaviors. See **os_Collection::create()** on page 64.

os_coll_rep_descriptor::copy()

os_coll_rep_descriptor &copy(os_segment*) const;

Creates a copy of the specified descriptor, allocated in the specified segment, and returns a reference to the copy.

os_coll_rep_descriptor::get_grow_rep_descriptor()

os_coll_rep_descriptor *get_grow_rep_descriptor() const;

Returns a pointer to the **os_coll_rep_descriptor** that becomes active when the cardinality increases past the growth threshold of the specified **os_coll_rep_descriptor**.

os_coll_rep_descriptor::get_max_size()

os_unsigned_int32 get_max_size() const;

Returns the maximum size of the specified descriptor.

os_coll_rep_descriptor::get_min_size()

os_unsigned_int32 get_min_size() const;

Returns the minimum size of the specified descriptor.

os_coll_rep_descriptor::rep_enum()

os_int32 rep_enum() const;

Returns an enumerator used to designate this representation type. See **os_Collection::create()** on page 64.

os_coll_rep_descriptor::rep_name()

char *rep_name() const;

Returns the name of this representation type. It is the user's responsibility to deallocate the returned string when it is no longer needed.

os_coll_rep_descriptor::required_behavior()

os_unsigned_int32 required_behavior() const;

Returns a bit-wise disjunction of enumerators indicating this representation's required behaviors. See **os_Collection::create()** on page 64.

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()**.

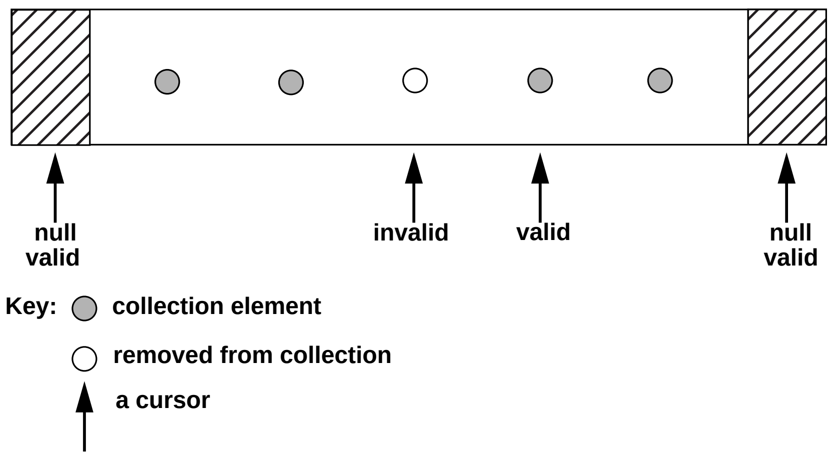
You can allocate a cursor in either transient or persistent memory.

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), the order in which elements appear in persistent memory (see **os_collection::order_by_address** on page 117), or an order based on an attribute or path of the elements. In the last case, the order is specified by an **os_index_path** specified when the cursor is created.

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::safe** was passed to the cursor constructor when the cursor was created, and the cursor's associated collection was created with **os_collection::maintain_cursors** behavior (see **os_collection::create()** on page 99). 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.



For a **safe** cursor whose associated collection maintains cursors, an invalid cursor's position is defined in terms of the immediate successor, **s**, of the removed element just prior to removal: such a cursor's position immediately after the removal is *between* **s** and the immediate predecessor, **p**, of **s**. This means performing **next()** on the cursor moves the cursor to **s**, and performing **previous()** moves the cursor to **p**.

If an invalid cursor is between an element, **s**, and the predecessor of **s**, **p**, and then elements are inserted between **p** and **s**, the cursor is then positioned between **s** and the new immediate predecessor, **p'**, of **s**.

In addition, whenever an invalid cursor is between an element, **s**, and its predecessor, **p**, removal of **s** results in repositioning the cursor so that it is between **p** and **s**'s immediate successor, and removal of **p** results in repositioning the cursor so that it is between **s** and **p**'s immediate predecessor.

A safe cursor whose associated collection maintains cursors has the following behavior during iteration:

- Any element that has been removed and not yet visited will not be visited.
- If the cursor's associated order is arbitrary, elements inserted during the iteration will be visited exactly once.

- If the iteration order was specified by an **os_index_path**, elements inserted before the current cursor position will not be visited, while those inserted after will be visited.

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 153.) This means that when specifying **os_Cursor** as a function's formal parameter, or as the type of a variable or data member, you must specify the parameter (the cursor's *element type*). This is accomplished by appending to **os_Cursor** the name of the element type enclosed in angle brackets, < >:

os_Cursor<*element-type-name*>

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.

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.

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 will be 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 will be 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 options = os_cursor::unsafe
);
```

Constructs a cursor associated with **coll**. If the collection is not ordered, the cursor's associated order is arbitrary, unless **option** is

os_collection::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 **option** is **os_cursor::unsafe** or **os_cursor::safe**, 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 or safe cursor, the results of the traversal are undefined.

If **option** is **os_collection::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 **option** is **os_collection::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 **option** is **os_cursor::unsafe**, the cursor does not support updates to its associated collection during iteration.

If **option** is **os_cursor::safe**, and the cursor's associated collection has the behavior specified by **os_collection::maintain_cursors**, the cursor supports updates during iteration over its associated collection. It visits any elements inserted later in the traversal order, and does not visit any elements that are later in the traversal order that are removed.

If **option** is **os_cursor::safe**, and the cursor's associated collection does not have the behavior specified by **os_collection::maintain_cursors**, err_coll_not_supported is signaled.

```
os_cursor(
    const os_collection & coll,
    _Rank_fcn rfcn,
    os_int32 options = os_cursor::unsafe
);
```

An **_Rank_fcn** is a rank function for the element type of **coll**. Iteration using that cursor will follow the order determined by the specified rank function. Rank-function-based cursors are update insensitive.

os_Cursor<E> (
const os_Collection<E> & coll,
const os_index_path &path,
os_int32 options = os_cursor::unsafe
);

Constructs a cursor associated with **coll**. The **path** specifies the cursor's associated order. If **safety** is **os_cursor::unsafe**, the cursor does not support updates to its associated collection during iteration. If safety is **os_cursor::safe**, and the cursor's associated collection has the behavior specified by **os_collection::maintain_cursors**, the cursor supports updates during iteration over its associated collection. If safety is **os_cursor::safe**, and the cursor's associated collection does not have the behavior specified by **os_collection::maintain_cursors**, err_coll_not_supported is signaled.

Upon creation of the first persistent, unsafe, ordered,or restricted cursor with a particular key type (where the key type is the specified path's terminal type), ObjectStore performs schema modification, provided the collection does not have an index on the specified path.

os_Cursor<E> (
const os_Collection<E> & coll,
const char *typename,
os_int32 options = os_cursor::unsafe
);

typename is the name of the element type. Iteration using that cursor will follow the order determined by the element type's rank function. Rank-function-based cursors are update insensitive.

os_Cursor<E> (
const os_Dictionary & coll,
const os_coll_range &range,
os_int32 options = os_cursor::unsafe
);

For traversing dictionaries. A traversal with this cursor visits only those collection elements whose key satisfies **range**. The order of iteration is arbitrary.

os_Cursor<E> (
const os_Collection<E> & coll,
const os_index_path &path,

```
    const os_coll_range &range,
    os_int32 options = os_cursor::unsafe
);
```

A traversal with this cursor visits only those collection elements that satisfy the cursor's restriction. An element satisfies the cursor's restriction if the result of applying **path** to the element satisfies **range**. The order of iteration is determined by **os_index_path** based on the index. If the index is not present, it is created.

You can construct a new cursor by copying with the following function, defined by the **os_cursor** class.

```
os_cursor (
    const os_cursor & c
);
```

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.

void rebind(const os_collection &, _Rank_Fcn);

Associates the specified cursor with the specified collection, positioning the cursor at the collection's first element.

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.

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), the order in which elements appear in persistent memory (see **os_collection::order_by_address** on page 117), or an order based on an attribute or path of the elements. In the last case, the order is specified by an **os_index_path** supplied when the cursor is created.

Upon creation of a persistent, unsafe, ordered cursor for which the collection does not have an index on the given path, a write lock is acquired on segment 0 that effectively locks the entire database.

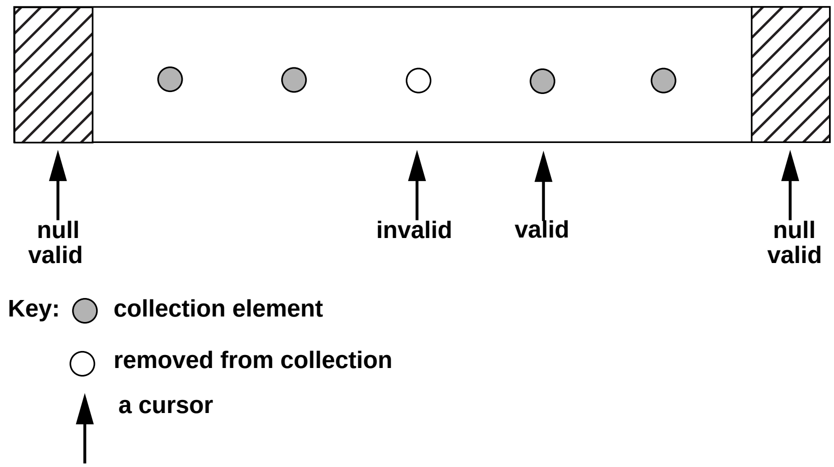
If a cursor is positioned at a collection's last element (in the cursor's associated ordering) and **next()** is performed on it, the cursor becomes *null*. Similarly, if a cursor is positioned at a collection's first element (in the cursor's associated ordering) and **previous()** is performed on it, 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_collection::safe** was passed to the cursor constructor when the cursor was created, and the cursor's associated collection was created with **maintain_cursors** behavior (see **os_collection::create()** on page 99). 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.



For **safe** cursors whose associated collection maintains cursors, an invalid cursor's position is defined in terms of the immediate successor, **s**, of the removed element just prior to removal: such a cursor's position immediately after the removal is *between* **s** and the immediate predecessor of **s**, **p**. This means performing **next()** on the cursor moves the cursor to **s**, and performing **previous()** moves the cursor to **p**.

If an invalid cursor is between an element, **s**, and its predecessor, **p**, and elements are subsequently inserted between **p** and **s**, the cursor is then positioned between **s** and the new immediate predecessor of **s**.

In addition, whenever an invalid cursor is between an element, **s**, and its predecessor, **p**, removal of **s** results in repositioning the cursor so that it is between **p** and **s**'s immediate successor. Similarly, removal of **p** results in repositioning the cursor so that it is between **s** and **p**'s immediate predecessor.

A safe cursor whose associated collection maintains cursors has the following behavior during iteration:

- Any element that has been removed and not yet visited will not be visited.
- If the cursor's associated order is arbitrary, elements inserted during the iteration will be visited exactly once.
- If the iteration order was specified by an **os_index_path**, elements inserted before the current cursor position will not be visited, while those inserted after will be visited.

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.

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.

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.

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::os_cursor()

```
os_cursor(
    const os_collection & coll,
    os_int32 options = os_cursor::unsafe
);
```

Constructs a cursor associated with **coll**. If the collection is not ordered, the cursor's associated order is system-supplied, unless **options** is **os_collection::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 **options** is **os_cursor::unsafe** or **os_cursor::safe**, 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 or safe cursor, the results of the traversal are undefined.

If **options** is **os_collection::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 **options** is **os_collection::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 **options** is **os_cursor::unsafe**, the cursor does not support updates to its associated collection during iteration.

If **options** is **os_cursor::safe**, and the cursor's associated collection has the behavior specified by **os_collection::maintain_cursors**, the cursor supports updates during iteration over its associated collection. It visits any elements inserted later in the traversal order, and does not visit any elements that are later in the traversal order that are removed.

If **options** is **os_cursor::safe**, and the cursor's associated collection does not have the behavior specified by **os_collection::maintain_cursors**, err_coll_not_supported is signaled.

```
os_cursor(
    const os_collection & coll,
    const os_index_path &path,
    os_int32 options = os_cursor::unsafe
);
```

Constructs a cursor associated with **coll**. The **path** specifies the cursor's associated order. If **options** is **os_cursor::unsafe**, the cursor does not support updates to its associated collection during iteration. If **options** is **os_cursor::safe**, and the cursor's associated collection has the behavior specified by **os_collection::maintain_cursors**, the cursor supports updates during iteration over its associated collection. If **options** is **os_cursor::safe**, and the cursor's associated collection does not have the behavior specified by **os_collection::maintain_cursors**, err_coll_not_supported is signaled.

Upon creation of the first persistent, unsafe, ordered, or restricted cursor with a particular key type (where the key type is the specified path's terminal type), provided the collection does not have an index on the specified path.

```
os_cursor(
    const os_collection & coll,
    const char *typename,
    os_int32 options = os_cursor::unsafe
);
```

typename is the name of the element type as argument. Iteration using that cursor will follow the order determined by the element type's rank function.

```
os_cursor(
    const os_collection & coll,
    _Rank_fcn rnk,
    os_int32 options = os_cursor::unsafe
);
```

An **_Rank_fcn** is a rank function for the element type of **coll**. Iteration using that cursor will follow the order determined by the specified rank function.

```
os_cursor(
    const os_collection & coll,
    const os_index_path &path,
    const os_coll_range &range,
    os_int32 options = os_cursor::unsafe
);
```

A traversal with this cursor visits only those collection elements that satisfy the cursor's restriction. An element satisfies the cursor's restriction if the result of applying **path** to the element satisfies **range**. The order of iteration is arbitrary.

Upon creation of a persistent, unsafe, ordered cursor for which the collection does not have an index on the given path, ObjectStore performs schema modification, which effectively write-locks the entire database.

```
os_cursor(
    const os_dictionary & coll,
    const os_coll_range &range,
    os_int32 options = os_cursor::unsafe
);
```

For traversing dictionaries. A traversal with this cursor visits only those collection elements whose key satisfies **range**. The order of iteration is arbitrary.

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(os_collection&);

Associates the specified cursor with the specified collection, positioning the cursor at the collection's first element.

void rebind(const os_collection&);

Associates the specified cursor with the specified collection, positioning the cursor at the collection's first element.

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.

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::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>

Like bags, dictionaries are unordered collections that allow duplicate elements. Unlike bags, however, dictionaries associate 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**.

os_rDictionary is just like **os_Dictionary**, except that it records its elements using references (as do **os_vdyn_hash** and **os_vdyn_bag**), which eliminates address space reservation and can reduce relocation overhead. See **os_rDictionary** on page 210 for a description of this class.

Dictionaries are always implemented as B-trees or hash tables, so lookup 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.

Create collections with the member **create()** or, for stack-based or embedded collections, with a constructor. Do not use **new** to create collections.

Requirements

Requirements for classes used as keys are listed below.

- Types used as keys also need a public **operator=.**
- 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)

- For class keys, the class must have a destructor, and if the class contains any pointers, they must be zeroed out.
- You must define and register (using **os_index_key**) rank / hash functions for the class type.

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 ordered, the dictionary makes its own copies of the character array upon insert. If the key type is **char*** and the dictionary has the behavior **maintain_key_order**, it will point to the string rather than making 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.

Although it is possible to set up an **os_Cursor** on an **os_Dictionary**, you cannot set up a safe cursor that allows insertions / removals during the iteration. That is, **os_Dictionary** does not support the behavior **os_collection::maintain_cursors**.

Required header files

Any program using dictionaries must include the header files **<ostore/ostore.hh>** followed by **<ostore/coll.hh>.** In addition your program will require the inclusion of **<ostore/coll/dict_pt.hh>** or **<ostore/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**.

Required libraries

Programs that use dictionaries must link with the library file **oscol.lib** (UNIX platforms) or **oscol.ldb** (Windows platforms).

Below are two tables. The first table 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*
add_index	**(const os_index_path&, os_int32 = unordered, os_segment* = 0)**	**void**	**os_collection**
	(const os_index_path&, os_int32 = unordered, os_database* = 0)	**void**	
	(const os_index_path&, os_segment* = 0)	**void**	
	(const os_index_path&, os_database* = 0)	**void**	
cardinality	**( ) const**	**os_unsigned_int32**	**os_collection**
change_behavior	**(os_unsigned_int32 behavior, os_int32 = verify)**	**void**	**os_collection**
clear	**( )**	**void**	**os_collection**
contains	**(const K &key_ref, const E element) const**	**os_int32**	**os_Dictionary**
	(const K *key_ptr, const E element) const	**os_int32**	
count	**(const E) const**	**os_int32**	**os_Collection**
count_values	**(const K &key_ref) const**	**os_int32**	**os_Dictionary**
	(const K * key_ptr) const	**os_unsigned_int32**	

Name	***Arguments***	***Returns***	***Defined By***
create (static)	**(os_database *db, os_unsigned_int32 expected_card = 10, os_unsigned_int32 behavior = 0)**	**os_Dictionary <K,E>&**	**os_Dictionary**
	(os_segment *seg, os_unsigned_int32 expected_card = 10, os_unsigned_int32 behavior = 0)	**os_Dictionary <K,E>&**	
	(os_object_cluster *clust, os_unsigned_int32 expected_card = 10, os_unsigned_int32 behavior = 0)	**os_Dictionary <K,E>&**	
	(os_object_cluster *proximity, os_unsigned_int32 expected_card = 10, os_unsigned_int32 behavior = 0)	**os_Dictionary <K,E>&**	
default_behavior (static)	**( )**	**os_unsigned_int32**	**os_Dictionary**
destroy (static)	**(os_Dictionary<K, E>&)**	**void**	**os_Dictionary**
drop_index	**(const os_index_path&)**	**void**	**os_collection**
empty	**( )**	**os_int32**	**os_collection**
exists	**(char *element_type_name, char *query_string, os_database *schema_database = 0, char *file, os_unsigned_int32 line) const**	**os_int32**	**os_collection**
	(const os_bound_query&) const	**os_int32**	
get_behavior	**( ) const**	**os_unsigned_int32**	**os_collection**
has_index	**(const os_index_path&, os_int32 index_options) const**	**os_int32**	**os_collection**
insert	**(const K &key_ref, const E element)**	**void**	**os_Dictionary**
	(const K *key_ptr, const E element)	**void**	
only	**( ) const**	**E**	**os_Collection**
os_Dictionary	**(os_unsigned_int32 expected_card = 10, os_unsigned_int32 behavior = 0)**		**os_Dictionary**
pick	**(const os_coll_range&) const**	**E**	**os_Dictionary**
	(const K &key_ref) const	**E**	
	(const K *key_ptr) const	**E**	
	( ) const	**E**	

Name	*Arguments*	*Returns*	*Defined By*
query	**(char *element_type_name, char *query_string, os_database *schema_database = 0, char *filename = 0, os_unsigned_int32 line = 0, os_boolean dups) const**	**os_Collection<E>&**	**os_Collection**
	(const os_bound_query&, os_boolean dups) const	**os_Collection<E>&**	
query_pick	**(char *element_type_name, char *query_string, os_database *schema_database = 0, char* filename = 0, os_unsigned_int32 line = 0) const**	**E**	**os_Dictionary**
	(const os_bound_query&) const	**E**	
remove	**(const K &key_ref, const E element)**	**void**	**os_Dictionary**
	(const K *key_ptr, const E element)	**void**	
remove_value	**(const K &key_ref, const E os_unsigned_int32 n = 1)**	**E**	**os_Dictionary**
	(const K *key_ptr, os_unsigned_int32 n = 1)	**E**	
retrieve	**(const os_cursor&) const**	**E**	**os_Dictionary**
retrieve_key	**(const os_cursor&)**	**K***	**os_Dictionary**

os_Dictionary enumerators

The following table lists enumerators for the **os_Dictionary** class.

Name	*Inherited From*
allow_nulls	**os_collection**
associate_policy	**os_collection**
dont_associate_policy	**os_collection**
dont_verify	**os_collection**
EQ	**os_collection**
GT	**os_collection**
LT	**os_collection**
maintain_cursors	**os_collection**
maintain_key_order	**os_Dictionary**
maintain_order	**os_collection**

Name	***Inherited From***
pick_from_empty_returns_null	**os_collection**
no_dup_keys	**os_Dictionary**
signal_cardinality	**os_collection**
signal_dup_keys	**os_Dictionary**
signal_duplicates	**os_collection**
unordered	**os_collection**
verify	**os_collection**

os_Dictionary::change_behavior()

change_behavior(os_unsigned_int_32 behavior);

Changes the behavior of the specified collection.

Take 1

behavior is a bit pattern, the bit-wise disjunction (using the operator |) of enumerators indicating all the desired properties for the changed collection. The enumerators are

- **os_collection::allow_nulls**
- **os_collection::signal_duplicates**
- **os_collection::signal_cardinality**
- **os_collection::pick_from_empty_returns_null**

When you change a collection so that it no longer allows null insertions, you might want to check to see if nulls are already present.

Take 2

You can customize the behavior of new dictionaries with regard to these last three properties. You do this by supplying a **behavior** argument to **create()**, an unsigned 32-bit integer, a bit pattern indicating the collection’s properties. The bit pattern is obtained by forming the bit-wise disjunction (using bit-wise or, **|**) of enumerators taken from the following possibilities:

- **os_collection::pick_from_empty_returns_null**: Performing **pick()** on an empty dictionary returns **0** rather than raising an exception.
- **os_dictionary::signal_dup_keys**: Duplicate keys are not allowed; err_am_dup_key is signaled if an attempt is made to establish two or more elements with the same key.

- **os_dictionary::maintain_key_order**: Range lookups are supported using **pick()** or restricted cursors.

These are instances of an enumeration defined in the scope of the **os_Dictionary**. Each enumerator is associated with a different bit, and including an enumerator in the disjunction sets its associated bit.

You can turn these behaviors on and off throughout the dictionary's lifetime. See **os_collection::change_behavior()** on page 97.

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 overloading following 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 preceding 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 overloading following 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 preceding overloading only in that the key is specified with a pointer instead of a reference.

os_Dictionary::create()

```
static os_Dictionary<K, E> &create(
    os_database *db,
    os_unsigned_int32 expected_cardinality = 10,
    os_unsigned_int32 behavior_enums = 0
);
```

Creates a new dictionary in the database pointed to by **db**. If the transient database is specified, the dictionary is allocated in transient memory. **K** can be a basic type, a pointer, or a class type. If the key type is a class type, the class's rank / hash functions must be registered with the **os_index_key** macro.

db: This is one of four overloadings of **create()**. As with the create operations for the other types of collections, these overloadings differ only in the first argument, which specifies where to allocate the new dictionary. Depending on the overloading, it specifies a database, segment, or object cluster.

expected_cardinality: Unlike the create operations for other collection classes, there are no arguments relating to representation policies. This is because you cannot directly control the representation for dictionaries.

By default, dictionaries are presized with a representation suitable for cardinality 10. If you want a new dictionary presized for a different cardinality, supply the **expected_cardinality** argument explicitly.

behavior: Every dictionary has the following properties:

- Its entries have no intrinsic order.
- 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:

- Performing **pick()** on an empty dictionary raises err_coll_empty.
- Duplicate keys are allowed; that is, two or more elements can have the same key.

- Range lookups are not supported; that is, key order is not maintained.

You can customize the behavior of new dictionaries with regard to these last three properties. You do this by supplying a **behavior** argument to **create()**, an unsigned 32-bit integer, a bit pattern indicating the collection's properties. The bit pattern is obtained by forming the bit-wise disjunction (using bit-wise or, **|**) of enumerators taken from the following possibilities:

- **os_collection::pick_from_empty_returns_null**: Performing **pick()** on an empty dictionary returns **0** rather than raising an exception.
- **os_dictionary::signal_dup_keys**: Duplicate keys are not allowed; err_am_dup_key is signaled if an attempt is made to establish two or more elements with the same key.
- **os_dictionary::maintain_key_order**: Range lookups are supported using **pick()** or restricted cursors.

These are instances of an enumeration defined in the scope of the **os_Dictionary**. Each enumerator is associated with a different bit, and including an enumerator in the disjunction sets its associated bit.

You can turn these behaviors on and off throughout the dictionary's lifetime. See **os_collection::change_behavior()** on page 97.

For large dictionaries that maintain key order, there is also an option for reducing contention. With **os_collection::dont_maintain_cardinality** behavior, **insert()** and **remove()** do not update cardinality information, avoiding contention in the collection header. This 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. See the following members of **os_collection()**: **os_collection::cardinality_is_maintained()** on page 97, **os_collection::cardinality_estimate()** on page 97, and **os_collection::update_cardinality()** on page 128.

Unlike the create operations for other collection classes, there are no arguments relating to representation. This is because you cannot directly control the representation for dictionaries. You can, however, use the class **os_rDictionary** instead of **os_ Dictionary**. The former class is just like **os_Dictionary**, except that it records its elements using references (as do **os_vdyn_hash** and **os_vdyn_bag**), which can eliminate address space reservation and can reduce relocation overhead. See the description of **os_ rDictionary** on page 210.

```
static os_Dictionary<K, E> &create(
    os_segment *seg,
    os_unsigned_int32 expected_cardinality = 10,
    os_unsigned_int32 behavior = 0
);
```

Creates a new dictionary in the segment pointed to by **seg**. If the transient segment is specified, the dictionary is allocated in transient memory.

This is one of four overloadings of **create()**. As with the create operations for the other types of collections, these overloadings differ only in the first argument, which specifies where to allocate the new dictionary. Depending on the overloading, it specifies a database, segment, or object cluster.

The rest of the arguments are just as described previously for the first overloading of this function.

```
static os_Dictionary<K, E> &create(
    os_object_cluster *clust,
    os_unsigned_int32 expected_cardinality = 10,
    os_unsigned_int32 behavior = 0
);
```

Creates a new dictionary in the object cluster pointed to by **clust**.

This is one of four overloadings of **create()**. As with the create operations for the other types of collections, these overloadings differ only in the first argument, which specifies where to allocate the new dictionary. Depending on the overloading, it specifies a database, segment, or object cluster.

The rest of the arguments are just as described previously for the first overloading of this function.

```
static os_Dictionary<K, E> &create(
    os_object_cluster *proximity,
    os_unsigned_int32 expected_cardinality = 10,
    os_unsigned_int32 behavior = 0
);
```

Creates a new dictionary in the segment occupied pointed to by **proximity**. If the object is part of an object cluster, the new dictionary is allocated in that cluster. If the specified object is transient, the array is allocated in transient memory. The rest of the arguments are just as described previously for the first overloading of this function.

os_Dictionary::default_behavior()

static unsigned long default_behavior ();

Returns a bit pattern indicating this type's default behavior, which includes allowing duplicates and allowing nulls. See the enumerators **os_collection::allow_duplicates** and **os_ collection::allow_nulls**.

os_Dictionary::destroy()

static void destroy(os_Dictionary<K, E>&);

Deletes the specified collection and deallocates associated storage. This is the same as calling **delete()** on the dictionary.

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 overloading following 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 insert 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 the same key as an element already present, 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 above overloading only in that the key is specified with a pointer instead of a reference. See the documentation for the preceding overloading.

os_Dictionary::os_Dictionary()

```
os_Dictionary(
    os_unsigned_int32 expected_cardinality = 10,
    os_unsigned_int32 behavior = 0
);
```

Use the dictionary constructor only to create stack-based dictionaries, or dictionaries embedded within other objects. See **os_Dictionary::create()**, above.

os_Dictionary::pick()

E pick(const os_coll_range&) const;

Returns an element of **this** that satisfies the specified **os_coll_range**. 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, err_coll_empty is signaled. If the dictionary has behavior **pick_from_empty_returns_null**, calling **os_Dictionary::pick()** on an empty dictionary returns **0**.

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, err_coll_empty is signaled. If the dictionary has behavior **pick_from_empty_returns_null**, calling **os_Dictionary::pick()** on an empty dictionary 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, err_coll_empty is signaled. If the dictionary has behavior **pick_from_empty_returns_null**, calling **os_Dictionary::pick()** on an empty dictionary returns **0**.

E pick() const;

Picks an arbitrary element of **this** and returns it. If the dictionary is empty, err_coll_empty is signaled, unless the collection's behavior includes **os_collection::pick_from_empty_returns_null**, in which case **0** is returned.

os_Dictionary::query()

???Returns an **os_collection**.

os_Dictionary::query_pick()

```
E query_pick(
    char *element_type,
    char *query_string,
    os_database *schema_database = 0,
    char *file_name = 0,
    os_unsigned_int32 line = 0
) const;
```

Returns an element of **this** that satisfies the specified **query_string**. See the documentation of **query_string** for **os_Collection::query()** on page 76.

If there is no such element or the dictionary is empty, **0** is returned.

The argument **element_type** is the name of the element type of **this**. Names established through the use of **typedef** are not allowed.

The **schema_database** is a database whose schema contains all the types mentioned in **query_string**. This database provides the environment in which the query is analyzed and optimized. The database in which the collection resides is often appropriate.

void *query_pick(const os_bound_query&) const;

Returns an element of **this** that satisfies the specified bound query. If there is no such element or the dictionary is empty, **0** is returned. If there is no such element and the dictionary does not have **pick_from_empty_returns_null** behavior, err_coll_empty is signaled.

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 referred to by **key_ref**. This overloading of **remove()** differs from the preceding 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. See the documentation for the previous overloading.

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 **insert()** differs from the preceding 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. See the documentation for the previous overloading.

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.

os_Dictionary::retrieve_key()

const K *retrieve_key(const os_cursor&) const;

Returns a pointer to a dictionary key. Do not modify the key being pointed to. Like all collections functions that take cursor arguments, this function works only with vanilla cursors. A vanilla cursor is any cursor that was not created with a cursor option, index path, rank function, or an **os_coll_range**.

os_dynamic_extent

Derived from **os_Collection**, an instance of this class can be used to create an extended collection of all objects of a particular type, regardless of which segments the objects reside in. All objects are retrieved in an arbitrary order that is stable across traversals of the segments, as long as no objects are created or deleted from the segment, and no reorganization is performed (using schema evolution or compaction).

os_dynamic_extent is useful for joining together multiple collections of the same object type into a new collection. The new collection is created dynamically, which results in no additional storage consumption.

You iterate over the **os_dynamic_extent** collection by creating an associated instance of **os_cursor**. Only the **os_cursor::more, os_cursor::first**, and **os_cursor::next** functions are supported by **os_dynamic_extent**. You can create an index for the **os_dynamic_extent** collection by calling **add_index()**; however, creating an index requires additional storage.

os_dynamic_extent::os_dynamic_extent()

```
os_dynamic_extent(
    os_database * db,
    os_typespec * typespec
);
```

Constructs an **os_dynamic_extent** that associates all objects of **os_typespec** that exist in the specified **os_database**. This constructor should be used only for transient instances of **os_dynamic_extent**.

```
os_dynamic_extent(
    os_typespec * typespec,
    os_boolean options = os_dynamic_extent::all_segments
);
```

Constructs an **os_dynamic_extent** that associates all objects of **os_typespec**. This constructor assumes that the **os_dynamic_extent** is persistent and searches the database where the **os_dynamic_extent** resides. If the option is **os_dynamic_extent::all_segments**, all segments are searched. The alternative option is **os_dynamic_extent::of_segment**, which searches only the segment in which the **os_dynamic_extent** is allocated.

```
os_dynamic_extent(
    os_database * db,
    os_typespec* typespec,
    os_segment* seg
);
```

Constructs an **os_dynamic_extent** that associates only those objects of **os_typespec** that exist in the specified **os_database** and o**s_segment**. This constructor should be used only for transient instances of **os_dynamic_extent.**

os_dynamic_extent::insert()

void insert(const void*);

Adds the specified **void*** to the index for the current **os_dynamic_extent** collection. You must first create an index by calling **os_dynamic_extent::add_index()**. See **os_collection::add_index()**.

os_dynamic_extent::remove()

os_int32 remove(const void*);

Removes the specified **void*** from the **os_dynamic_extent** collection index.

If the index is ordered, the first occurrence of the specified **void*** is removed. Returns a nonzero **os_int32** if an element was removed; returns **0** otherwise.

os_dynamic_extent::~os_dynamic_extent()

~os_dynamic_extent();

Performs internal maintenance associated with **os_dynamic_extent** deallocation.

os_index_name

An instance of this class encapsulates information about a particular index. Functions are provided for retrieving a string representation of the index's associated path and a bit pattern indicating the index's associated options. See also **os_ collection::get_indexes()** on page 105, which returns a collection of **os_index_name**s.

os_index_name::get_options()

os_int32 get_options();

Returns a bit pattern indicating the index options associated with the index named by **this**.

os_index_name::get_path_name()

char *get_path_name();

Returns a string representation of the path associated with the index named by **this**. The caller is responsible for freeing the memory pointed to by the return value.

os_index_path

Instances of the class **os_index_path** are used in specifying iteration order, as well as in specifying index keys to enable query optimization.

Each path specifies a certain kind of mapping by specifying a sequence of member names. Applying the mapping is equivalent to accessing a data member, applying a member function, or accessing a data member of a data member, and so on.

For example, suppose the type **employee** defines a data member **department**, whose values are pointers to instances of a type that defines a data member **manager**. Then a path might be specified with the **path_string "department–>manager"**. This path maps pointers to instances of **employee** to the manager of the given employee's department.

Path expressions have some additional expressive power. They can indicate iterative retrieval of the elements of path results that are collections. For example, consider specifying a key for an index that optimizes lookup of a part based on the **emp_id** of any of the **responsible_engineers** for the part (suppose that the member **responsible_engineers** is collection valued). You can use the path created by the following call:

```
os_index_path::create(
    "part*","responsible_engineers[]->emp_id", db1)
```

Here the data member name "**responsible_engineers**" is followed by the symbols **[ ]**, indicating that the next component of the path (**emp_id**) is to be applied to *each element* of the collection of responsible engineers, rather than to the collection itself.

os_index_path::create()

```
static os_index_path &create(
    const char *element_type_string,
    const char *path_string,
    const os_database*
);
```

Creates a transient heap-allocated **os_index_path**.

The **element_type_string**, known as the path's *type string*, is a string consisting of the name of the element type of collections whose elements can serve as path starting points. Names created with a typedef cannot be used.

The **path_string** consists of a sequence of member names, separated by dots or arrows.

Given a path string, **path-string**, ending in a member function name, you can form a path string whose values are the results of dereferencing the result of **path-string** this way:

" *(parent-> theChild()) "

***(*path-string*)**

The parentheses are not necessary if the original path string specifies a single-step path.

You cannot specify a dereferenced data member at the end of a path string. For example, you cannot specify

" *(parent->child) "

or

" * Foo "

A collection-valued path followed by a pair of brackets **[ ]** forms a multivalued path whose values are the collection's elements. To indicate element retrieval (using **[]**) from a nonparameterized collection, the part of the path designating the collection must be preceded by a cast to **os_Collection<E>**, where **E** is the collection's element type **os_collection**.

The value type of a data member referred to in a path expression must be a built-in type, a class, or a pointer to a built-in type or class. The type **person****, for example, is not allowed.

If an illegal **path_string** is supplied, err_illegal_index_path is signaled.

Data members mentioned in the **path_string**, except **const** and collection-valued members, must be indexable if there is any possibility of their being updated when participating in an index.

For applications that use member functions, see "Member function in a query string" on page 78. In addition, when you create an index path that ends in a member function, the member function should return something that is either a basic type, a pointer to a persistent object, or a class object that will be copied into the index.

An application must supply a rank function (see Chapter 5, User-Supplied Functions, in *ObjectStore C++ API Reference*) for a class type, **T**, if the application uses a path ending in **T** as an index key or to specify iteration order. An application must supply a hash function (see Chapter 5, User-Supplied Functions, in *ObjectStore C++ API Reference*) for a class, **T**, if the application uses a path ending in **T** as a key for an unordered index.

The **os_database*** is a database whose schema contains the classes defining the members mentioned in the **path_string**.

Once the path generated by **create()** is no longer needed, you should deallocate with **::operator delete()** to avoid memory leaks.

```
static os_index_path &create(
    const char *element_type_string,
    const char *path_string,
    const os_segment*
);
```

Creates a transient heap-allocated **os_index_path**. Same as the preceding version of **create()**, except that the **os_segment*** indicates a segment in a database whose schema contains the classes defining the members mentioned in the **path_string**.

```
static os_index_path &create(
    const char *element_type_string,
    const char *path_string,
    const void*
);
```

Creates a transient heap-allocated **os_index_path**. Same as the preceding version of **create()**, except that the **void*** indicates an object in a database whose schema contains the classes defining the members mentioned in the **path_string**.

os_index_path::destroy()

```
static void destroy(os_index_path&);
```

Deletes the specified path. This is the same as deleting the **os_index_path**.

os_keyword_arg

An instance of this class is used to specify the binding of a free reference in an **os_coll_query**. An **os_keyword_arg** or **os_keyword_arg_list** (see **os_keyword_arg_list** on page 185) is used together with an **os_coll_query** to create an **os_bound_query**. Each **os_keyword_arg** associates a variable name with a value of the appropriate type.

os_keyword_arg::operator ,()

os_keyword_arg_list &operator ,(const os_keyword_arg arg&)const;

Returns a reference to an **os_keyword_arg_list** whose elements are the instances of **os_keyword_arg** referred to by **this** and **arg**. The comma operator of this class and **os_keyword_arg_list** is overloaded in such a way that you can designate a **keyword_arg_list** with an expression of the following form:

```
(
    keyword_arg-expr,
    keyword_arg-expr,
    ... ,
    keyword_arg-expr
)
```

os_keyword_arg::os_keyword_arg()

```
os_keyword_arg(
    const char *name,
    os_signed_int8 value
);
```

Constructs an **os_keyword_arg** that binds the **char-**valued variable specified by **name** to **value**.

```
os_keyword_arg(
    const char *name ,
    unsigned char value
);
```

Constructs an **os_keyword_arg** that binds the **unsigned char-**valued variable specified by **name** to **value**.

```
os_keyword_arg(
    const char *name,
    short value
);
```

Constructs an **os_keyword_arg** that binds the **short-**valued variable specified by **name** to **value**.

```
os_keyword_arg(
    const char *name,
    unsigned short value
);
```

Constructs an **os_keyword_arg** that binds the **unsigned short-**valued variable specified by **name** to **value**.

```
os_keyword_arg(
    const char *name,
    int value
);
```

Constructs an **os_keyword_arg** that binds the **int-**valued variable specified by **name** to **value**.

```
os_keyword_arg(
    const char *name,
    unsigned int value
);
```

Constructs an **os_keyword_arg** that binds the **unsigned int-**valued variable specified by **name** to **value**.

```
os_keyword_arg(
    const char *name,
    long value
);
```

Constructs an **os_keyword_arg** that binds the **long-**valued variable specified by **name** to **value**.

```
os_keyword_arg(
    const char *name,
    unsigned long value
);
```

Constructs an **os_keyword_arg** that binds the **unsigned long-**valued variable specified by **name** to **value**.

```
os_keyword_arg(
    const char *name,
    float value
);
```

Constructs an **os_keyword_arg** that binds the **float-**valued variable specified by **name** to **value**.

```
os_keyword_arg(
    const char *name,
```

```
    double value
);
```

Constructs an **os_keyword_arg** that binds the **double-**valued variable specified by **name** to **value**.

```
os_keyword_arg(
    const char *name,
    long double value
);
```

Constructs an **os_keyword_arg** that binds the **long double-**valued variable specified by **name** to **value**.

```
os_keyword_arg(
    const char *name,
    void* value
);
```

Constructs an **os_keyword_arg** that binds the **void*-**valued variable specified by **name** to **value** and **void***.

```
os_keyword_arg(
    const char *name,
    const void* value
);
```

Constructs an **os_keyword_arg** that binds the **const void*-**valued variable specified by **name** to **value** and **void***.

os_keyword_arg_list

An instance of this class is used to specify the binding of free variables in an **os_coll_query**. An **os_keyword_arg_list** or **os_keyword_arg** (see **os_keyword_arg** on page 182) is used together with an **os_coll_query** to create an **os_bound_query**. Each **os_keyword_arg_list** associates a variable name with a value of the appropriate type.

os_keyword_arg_list::operator ,()

os_keyword_arg_list &operator ,(const os_keyword_arg&);

Returns a reference to an **os_keyword_arg_list** whose elements are the elements of **this** together with the specified **os_keyword_arg**. The comma operator of this class and **os_keyword_arg** is overloaded in such a way that you can designate an **os_keyword_arg_list** with an expression of the following form:

(
keyword_arg-expr,
keyword_arg-expr,
... ,
keyword_arg-expr
)

os_keyword_arg_list::os_keyword_arg_list()

```
os_keyword_arg_list(
    const os_keyword_arg&,
    const os_keyword_arg_list* = 0
);
```

Constructs an **os_keyword_arg_list** whose elements are the specified **os_keyword_arg** together with the elements of the specified **os_keyword_arg_list**. This constructor allows conversion of an **os_keyword_arg** to a single-element **os_keyword_arg_list**. This is useful when calling the **os_bound_query** constructor. For example:

```
os_bound_query bq(my_coll_query, os_keyword_arg(age, 5));
```

Between this constructor, the comma operator of this class, and the comma operator of **os_keyword_arg**, it should never be necessary for you to reference an **os_keyword_arg_list** explicitly.

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.

By default, lists are ordered, 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 198). This means that when specifying **os_List** as a function's formal parameter, or as the type of a variable or data member, you must specify the parameter (the list's *element type).* This is accomplished by appending to **os_List** the name of the element type enclosed in angle brackets, < >:

os_List<*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_List** must be a pointer type.

Create collections with the member **create()** or, for stack-based or embedded collections, with a constructor. Do not use **new** to create collections.

Required header files

Programs that use lists must include the header file **<ostore/coll.hh>** after including **<ostore/ostore.hh>**.

Required libraries

Programs that use lists must link with the library files **liboscol.so** and **liboscol.ldb** (UNIX platforms) or **oscol.ldb** and **oscoll.lib** (Windows platforms).

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.

Below are two tables. The first table 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*
add_index	**(const os_index_path&, os_int32 = unordered, os_segment* = 0)**	**void**	**os_collection**
	(const os_index_path&, os_int32 = unordered, os_database* = 0)	**void**	
	(const os_index_path&, os_segment* = 0)	**void**	
	(const os_index_path&, os_database* = 0)	**void**	
cardinality	**( ) const**	**os_int32**	**os_collection**
change_behavior	**(os_unsigned_int32 behavior, os_int32 = verify)**	**void**	**os_collection**
change_rep	**(os_unsigned_int32 expected_size, const os_coll_rep_descriptor *policy = 0, os_int32 retain = dont_associate_policy)**	**void**	**os_collection**
clear	**( )**	**void**	**os_collection**
contains	**(const E) const**	**os_int32**	**os_Collection**
count	**(const E) const**	**os_int32**	**os_Collection**
create (static)	**(os_database *db, os_unsigned_int32 behavior = 0, os_int32 expected_size = 0, const os_coll_rep_descriptor* = 0, os_int32 retain = dont_associate_policy)**	**os_List<E>&**	**os_List**

Name	*Arguments*	*Returns*	*Defined By*
	(os_segment *seg, os_unsigned_int32 behavior = 0, os_int32 expected_size = 0, const os_coll_rep_descriptor* = 0, os_int32 retain = dont_associate_policy)	**os_List<E>&**	
	(os_object_cluster *clust, os_unsigned_int32 behavior = 0, os_int32 expected_size = 0, const os_coll_rep_descriptor* = 0, os_int32 retain = dont_associate_policy)	**os_List<E>&**	
	(void* proximity, os_unsigned_int32 behavior = 0, os_int32 expected_size = 0, const os_coll_rep_descriptor* = 0, os_int32 retain = dont_associate_policy)	**os_List<E>&**	
default_behavior (static)	**( )**	**os_unsigned_int32**	**os_List**
destroy (static)	**(os_List<E>&)**	**void**	**os_List**
drop_index	**(const os_index_path&)**	**void**	**os_collection**
empty	**( )**	**os_int32**	**os_collection**
exists	**(char *element_type_name, char *query_string, os_database *schema_database = 0, char* file, os_unsigned_int32 line) const**	**os_int32**	**os_collection**
	(const os_bound_query&) const	**os_int32**	
get_behavior	**( ) const**	**os_unsigned_int32**	**os_collection**
get_rep	**( ) const**	**os_coll_rep_ descriptor&**	**os_collection**
has_index	**(const os_index_path&, os_int32 index_options = unordered) const**	**os_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>&)**	**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<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**	**os_int32**	**os_Collection**
	(const E) const	**os_int32**	
operator !=	**(const os_Collection<E>&) const**	**os_int32**	**os_Collection**
	(const E) const	**os_int32**	

<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</td><td>os_int32</td><td>os_Collection</td></tr>
<tr><td></td><td>(const E) const</td><td>os_int32</td><td></td></tr>
<tr><td>operator <=</td><td>(const os_Collection<E>&) const</td><td>os_int32</td><td>os_Collection</td></tr>
<tr><td></td><td>(const E) const</td><td>os_int32</td><td></td></tr>
<tr><td>operator ></td><td>(const os_Collection<E>&) const</td><td>os_int32</td><td>os_Collection</td></tr>
<tr><td></td><td>(const E) const</td><td>os_int32</td><td></td></tr>
<tr><td>operator >=</td><td>(const os_Collection<E>&) const</td><td>os_int32</td><td>os_Collection</td></tr>
<tr><td></td><td>(const E) const</td><td>os_int32</td><td></td></tr>
<tr><td>operator =</td><td>(const os_List<E>&) const</td><td>os_List<E>&</td><td>os_List</td></tr>
<tr><td></td><td>(const os_Collection<E>&) const</td><td>os_List<E>&</td><td></td></tr>
<tr><td></td><td>(const E) const</td><td>os_List<E>&</td><td></td></tr>
<tr><td>operator |=</td><td>(const os_Collection<E>&) const</td><td>os_List<E>&</td><td>os_List</td></tr>
<tr><td></td><td>(const E) const</td><td>os_List<E>&</td><td></td></tr>
<tr><td>operator |</td><td>(const os_Collection<E>&) const</td><td>os_Collection<E>&</td><td>os_Collection</td></tr>
<tr><td></td><td>(const E) const</td><td>os_Collection<E>&</td><td></td></tr>
<tr><td>operator &=</td><td>(const os_Collection<E>&) const</td><td>os_List<E>&</td><td>os_List</td></tr>
<tr><td></td><td>(const E) const</td><td>os_List<E>&</td><td></td></tr>
<tr><td>operator &</td><td>(const os_Collection<E>&) const</td><td>os_Collection<E>&</td><td>os_Collection</td></tr>
<tr><td></td><td>(const E) const</td><td>os_Collection<E>&</td><td></td></tr>
<tr><td>operator –=</td><td>(const os_Collection<E>&) const</td><td>os_List<E>&</td><td>os_List</td></tr>
<tr><td></td><td>(const E) const</td><td>os_List<E>&</td><td></td></tr>
<tr><td>operator -</td><td>(const os_Collection<E>&) const</td><td>os_Collection<E>&</td><td>os_Collection</td></tr>
<tr><td></td><td>(const E) const</td><td>os_Collection<E>&</td><td></td></tr>
<tr><td>os_List</td><td>( )</td><td></td><td>os_List</td></tr>
<tr><td></td><td>(os_collection_size expected_size)</td><td></td><td></td></tr>
<tr><td></td><td>(const os_List<E>&)</td><td></td><td></td></tr>
<tr><td></td><td>(const os_Collection<E>&)</td><td></td><td></td></tr>
<tr><td>pick</td><td>( ) const</td><td>E</td><td>os_Collection</td></tr>
<tr><td></td><td>(const os_index_path&,
const os_coll_range&) const</td><td>E</td><td></td></tr>
</table>

Name	*Arguments*	*Returns*	*Defined By*
query	**(char *element_type_name, char *query_string, os_database *schema_database = 0, char* filename = 0, os_unsigned_int32 line, os_boolean dups) const**	**os_Collection<E>&**	**os_Collection**
	(const os_bound_query&, os_boolean dups) const	**os_Collection<E>&**	
query_pick	**(char *element_type_name, char *query_string, os_database *schema_database = 0, char* file, os_unsigned_int32 line) const**	**E**	**os_Collection**
	(const os_bound_query&) const	**E**	
remove	**(const E)**	**os_int32**	**os_Collection**
remove_at	**(const os_Cursor<E>&)**	**void**	**os_Collection**
	(os_unsigned_int32)	**void**	
remove_first	**(const E&)**	**os_int32**	**os_Collection**
	( )	**E**	
remove_last	**(const E&)**	**os_int32**	**os_Collection**
	( )	**E**	
replace_at	**(const E, const os_Cursor<E>&)**	**E**	**os_Collection**
	(const E, os_unsigned_int32)	**E**	
retrieve	**(os_unsigned_int32) const**	**E**	**os_Collection**
	(const os_Cursor<E>&) const	**E**	
retrieve_first	**( ) const**	**E**	**os_Collection**
	(const E&) const	**os_int32**	
retrieve_last	**( ) const**	**E**	**os_Collection**
	(const E&) const	**os_int32**	

os_List enumerators

The following table lists the enumerators inherited by **os_List** from **os_collection**.

Name	*Inherited From*
allow_duplicates	**os_collection**

Name	***Inherited From***
allow_nulls	**os_collection**
associate_policy	**os_collection**
dont_associate_policy	**os_collection**
dont_verify	**os_collection**
EQ	**os_collection**
GT	**os_collection**
LT	**os_collection**
maintain_cursors	**os_collection**
maintain_order	**os_collection**
order_by_address	**os_collection**
pick_from_empty_returns_null	**os_collection**
signal_cardinality	**os_collection**
signal_duplicates	**os_collection**
unordered	**os_collection**
verify	**os_collection**

os_List::create()

```
static os_List<E> &create (
    os_database *db,
    os_unsigned_int32 behavior = 0,
    os_int32 expected_size = 0,
    const os_coll_rep_descriptor *rep_policy = 0,
    os_int32 retain = dont_associate_policy
);
```

Creates a new list in the database pointed to by **db**. If the transient database is specified, the list is allocated in transient memory.

The **behavior** is a bit pattern, the bit-wise disjunction (using the operator |) of enumerators indicating the desired behaviors. The enumerators are

- **os_collection::allow_nulls**
- **os_collection::allow_duplicates**

- **os_collection::signal_duplicates**
- **os_collection::pick_from_empty_returns_null**
- **os_collection::maintain_cursors**
- **os_collection::be_an_array**

See the class **os_collection** on page 87 for an explanation of each enumerator.

The specified behaviors supplement the default behaviors for lists. If 0 is supplied for **behavior**, only the default behaviors are enabled. The default behavior is given by

os_collection::maintain_order | os_collection::allow_duplicates

A run-time error is signaled if an attempt is made to create a list that is not ordered.

Representation policy

The default representation policy for lists created with **create()** is as follows:

- A list created as an embedded object has the representation of **os_tiny_array** (0 to 4 elements).
- An embedded list becomes out of line and mutates to an **os_chained_list** when the fifth element is inserted.
- A list created with **::create** with cardinality <= 20 is represented as an **os_chained_list**.
- Once the list grows past 20, its representation is **os_packed_list**.

The **expected_size** is the cardinality you expect the collection to have when fully loaded. This value is used by ObjectStore to determine the collection's initial representation. This saves on the overhead of transforming the collection's representation as it grows during loading.

The **rep_policy** is the representation policy to be associated with the collection until explicitly changed, if **retain** is **os_collection::associate_policy**. If **retain** is **os_collection::dont_associate_policy**, the **rep_policy** is used, together with the **expected_size**, only to determine the collection's initial representation. (A representation policy is, essentially, a mapping from cardinality ranges to representation types — see **os_coll_rep_descriptor** on page 143, and in *ObjectStore Advanced C++ API User Guide* see **os_ptr_bag** and **os_dyn_bag**.)

Additional behaviors

An **os_List** can also have these behaviors:

- **pick_from_empty_returns_null**
- **signal_duplicates**
- **allow_nulls**
- **maintain_cursors**

```
static os_List<E> &create (
    os_segment * seg,
    os_unsigned_int32 behavior = 0,
    os_int32 expected_size = 0,
    const os_coll_rep_descriptor *rep_policy = 0,
    os_int32 retain = dont_associate_policy
);
```

Creates a new list in the segment pointed to by **seg**. If the transient segment is specified, the list is allocated in transient memory. The rest of the arguments are just as described previously.

```
static os_List<E> &create (
    os_object_cluster *clust,
    os_unsigned_int32 behavior = 0,
    os_int32 expected_size = 0,
    const os_coll_rep_descriptor *rep_policy = 0,
    os_int32 retain = dont_associate_policy
);
```

Creates a new list in the object cluster pointed to by **clust**. The rest of the arguments are just as described previously.

```
static os_List<E> &create(
    void * proximity,
    os_unsigned_int32 behavior = 0,
    os_int32 expected_size = 0,
    const os_coll_rep_descriptor *rep_policy = 0,
    os_int32 retain = dont_associate_policy
);
```

Creates a new list in the segment occupied by the object pointed to by **proximity**. If the object is part of an object cluster, the new list is allocated in that cluster. If the specified object is transient, the list is allocated in transient memory. The rest of the arguments are just as described previously.

os_List::default_behavior()

```
static os_unsigned_int32 default_behavior();
```

Returns a bit pattern indicating this type's default behavior, which is **maintain_order** and **allow_duplicates**.

os_List::destroy()

static void destroy(os_List<E>&);

Deletes the specified collection and deallocates associated storage. This is the same as deleting the list.

Assignment Operator Semantics

Note: The assignment operator semantics are described in the next section in terms of insert operations into the target collection. Describing the semantics in terms of insert operations serves to illustrate how duplicate, null, and order semantics are enforced. The actual implementation of the assignment might be quite 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_collection_size);

The user should pass an **os_int32** for the **os_collection_size** actual argument. Returns an empty list whose initial implementation is based on the expectation that the specified **os_int32** indicates the approximate usual cardinality of the list, once it has been loaded with elements.

os_List(const os_List<E>&);

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

os_List(const os_Collection<E>&);

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 186.

By default, lists allow duplicates and disallow null elements.

The element type of any instance of **os_list** must be a pointer type.

Create collections with the member **create()** or, for stack-based or embedded collections, with a constructor. Do not use **new** to create collections.

Behavior

- The **behavior** is a bit pattern, the bit-wise disjunction (using the operator |) of enumerators indicating the desired properties. The enumerators are
- **os_collection::allow_nulls**
- **os_collection::allow_duplicates**
- **os_collection::signal_duplicates**
- **os_collection::pick_from_empty_returns_null**
- **os_collection::maintain_cursors**
- **os_collection::be_an_array**

See the class **os_collection** on page 87 for an explanation of each enumerator.

The specified behaviors supplement the default behaviors for lists. If 0 is supplied for **behavior**, only the default behaviors are enabled. The default behavior is given by

os_collection::maintain_order | os_collection::allow_duplicates

A run-time error is signaled if an attempt is made to create a list that is not ordered.

Expected cardinality

The **expected_size** is the cardinality you expect the collection to have when fully loaded. This value is used by ObjectStore to determine the collection's initial representation. This saves on the overhead of transforming the collection’s representation as it grows during loading.

Representations

The **rep_policy** is the representation policy to be associated with the collection until explicitly changed, if **retain** is **os_collection::associate_policy**. If **retain** is **os_collection::dont_associate_policy**, the **rep_policy** is used, together with the **expected_size**, only to determine the collection’s initial representation. (A representation policy is, essentially, a mapping from cardinality ranges to representation types — see **os_coll_rep_descriptor** on page 143, and in *ObjectStore Advanced C++ API User Guide* see **os_ptr_bag** and **os_dyn_bag**.)

Required header files

Programs that use lists must include the header file **<ostore/coll.hh>** after including **<ostore/ostore.hh>**.

Required libraries

Programs that use lists must link with the library file **oscol.lib** (UNIX platforms) or **oscol.ldb** (Windows platforms).

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.

Tables of member functions and enumerators

Below are two tables. The first table 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*
add_index	**(const os_index_path&, os_int32 = unordered, os_segment* = 0)**	**void**	**os_collection**
	(const os_index_path&, os_int32 = unordered, os_database* = 0)	**void**	
	(const os_index_path&, os_segment* = 0)	**void**	
	(const os_index_path&, os_database* = 0)	**void**	

Name	*Arguments*	*Returns*	*Defined By*
cardinality	**( ) const**	**os_int32**	**os_collection**
change_behavior	**(os_unsigned_int32 behavior, os_int32 = verify)**	**void**	**os_collection**
change_rep	**(os_unsigned_int32 expected_size, const os_coll_rep_descriptor *policy = 0, os_int32 retain = dont_associate_policy)**	**void**	**os_collection**
clear	**( )**	**void**	**os_collection**
contains	**(const void*) const**		**os_collection**
count	**(const void*) const**	**os_int32**	
create (static)	**(os_segment *seg, os_unsigned_int32 behavior = 0, os_int32 expected_size = 0, const os_coll_rep_descriptor* = 0, os_int32 retain = dont_associate_policy)**	**os_list**	**os_list**
	(os_database *db, os_unsigned_int32 behavior = 0, os_int32 expected_size = 0, const os_coll_rep_descriptor* = 0, os_int32 retain = dont_associate_policy)	**os_list**	
	(os_object_cluster *clust, os_unsigned_int32 behavior = 0, os_int32 expected_size = 0, const os_coll_rep_descriptor* = 0, os_int32 retain = dont_associate_policy)	**os_list**	
	(void* proximity, os_unsigned_int32 behavior = 0, os_int32 expected_size = 0, const os_coll_rep_descriptor* = 0, os_int32 retain = dont_associate_policy)	**os_list**	
default_behavior (static)	**( )**	**os_unsigned_int32**	**os_set**
destroy (static)	**(os_list&)**	**void**	**os_list**
drop_index	**(const os_index_path&)**	**void**	**os_collection**
empty	**( )**	**os_int32**	**os_collection**
exists	**(char *element_type_name, char *query_string, os_database *schema_database = 0, char* file, os_unsigned_int32 line) const**	**os_int32**	**os_collection**
	(const os_bound_query&) const	**os_int32**	

Name	*Arguments*	*Returns*	*Defined By*
get_behavior	**( ) const**	**os_unsigned_int32**	**os_collection**
get_rep	**( ) const**	**os_coll_rep_ descriptor&**	**os_collection**
has_index	**(const os_index_path&, os_int32 index_options = unordered) const**	**os_int32**	**os_collection**
insert	**(const void*)**	**void**	**os_collection**
insert_after	**(const void*, const os_cursor&)**	**void**	**os_collection**
	(const void*, os_unsigned_int32)	**void**	
insert_before	**(const void*, const os_cursor&)**	**void**	**os_collection**
	(const void*, os_unsigned_int32)	**void**	
insert_first	**(const void*)**	**void**	**os_Collection**
insert_last	**(const void*)**	**void**	**os_Collection**
only	**( ) const**	**void***	**os_Collection**
operator os_bag&	**( )**		**os_collection**
operator const os_bag&	**( ) const**		**os_collection**
operator os_set&	**( )**		**os_collection**
operator const os_set&	**( ) const**		**os_collection**
operator ==	**(const os_collection&) const**	**os_int32**	**os_collection**
	(const void*) const	**os_int32**	
operator !=	**(const os_collection&) const**	**os_int32**	**os_collection**
	(const void*) const	**os_int32**	
operator <	**(const os_collection&) const**	**os_int32**	**os_collection**
	(const void*) const	**os_int32**	
operator <=	**(const os_collection&) const**	**os_int32**	**os_collection**
	(const void*) const	**os_int32**	
operator >	**(const os_collection&) const**	**os_int32**	**os_collection**
	(const void*) const	**os_int32**	

Name	*Arguments*	*Returns*	*Defined By*
operator >=	**(const os_collection&) const**	**os_int32**	**os_collection**
	(const void*) const	**os_int32**	
operator =	**(const os_list&) const**	**os_list&**	**os_list**
	(const os_collection&) const	**os_list&**	
	(const void*) const	**os_list&**	
operator |=	**(const os_collection&) const**	**os_list&**	**os_list**
	(const void*) const	**os_list&**	
operator |	**(const os_collection&) const**	**os_collection**	**os_list**
	(const void*) const	**os_collection**	
operator &=	**(const os_collection&) const**	**os_list&**	**os_list**
	(const void*) const	**os_list&**	
operator &	**(const os_collection&) const**	**os_collection**	**os_list**
	(const void*) const	**os_collection**	
operator –=	**(const os_collection&) const**	**os_list&**	**os_list**
	(const void*) const	**os_list&**	
operator -	**(const os_collection&) const**	**os_collection**	**os_list**
	(const void*) const	**os_collection**	
os_list	**( )**		**os_list**
	(os_collection_size expected_size)		
	(const os_list&)		
	(const os_collection&)		
pick	**( ) const**	**void***	**os_collection**
	(const os_index_path&, const os_coll_range&) const	**void***	
query	**(char *element_type_name, char *query_string, os_database *schema_database = 0, char* filename = 0, os_unsigned_int32 line, os_boolean dups) const**	**os_collection&**	**os_collection**
	(const os_bound_query&, os_boolean dups) const	**os_collection&**	

Name	*Arguments*	*Returns*	*Defined By*
query_pick	**(char *element_type_name, char *query_string, os_database *schema_database = 0, char* file, os_unsigned_int32 line) const**	**void***	**os_collection**
	(const os_bound_query&) const	**void***	
remove	**(const void*)**	**os_int32**	**os_collection**
remove_at	**(const os_cursor&)**	**void**	**os_collection**
	(os_unsigned_int32)	**void**	
remove_first	**(const void*&)**	**os_int32**	**os_collection**
	( )	**void***	
remove_last	**(const void*&)**	**os_int32**	**os_collection**
	( )	**void***	
replace_at	**(const void*, const os_cursor&)**	**void***	**os_collection**
	(const void*, os_unsigned_int32)	**void***	
retrieve	**(os_unsigned_int32) const**	**void***	**os_collection**
	(const os_cursor&) const	**void***	
retrieve_first	**( ) const**	**void***	**os_collection**
	(const void*&) const	**os_int32**	
retrieve_last	**( ) const**	**void***	**os_collection**
	(const void*&) const	**os_int32**	

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**
associate_policy	**os_collection**
dont_associate_policy	**os_collection**
dont_verify	**os_collection**
EQ	**os_collection**
GT	**os_collection**

Name	Inherited From
LT	os_collection
maintain_cursors	os_collection
maintain_order	os_collection
order_by_address	os_collection
pick_from_empty_returns_null	os_collection
signal_cardinality	os_collection
signal_duplicates	os_collection
unordered	os_collection
verify	os_collection

os_list::create()

```
static os_list &create (
    os_database *db,
    os_unsigned_int32 behavior = 0,
    os_int32 expected_size = 0,
    const os_coll_rep_descriptor *rep_policy = 0,
    os_int32 retain = dont_associate_policy
);
```

Creates a new list in the database pointed to by **db**. If the transient database is specified, the list is allocated in transient memory.

The **behavior** is a bit pattern, the bit-wise disjunction (using the operator |) of enumerators indicating the desired properties. The enumerators are

- **os_collection::allow_nulls**
- **os_collection::allow_duplicates**
- **os_collection::signal_duplicates**
- **os_collection::pick_from_empty_returns_null**
- **os_collection::maintain_cursors**
- **os_collection::be_an_array**

See the class **os_collection** on page 87 for an explanation of each enumerator.

The specified behaviors supplement the default behaviors for lists. If 0 is supplied for **behavior**, only the default behaviors are enabled. The default behavior is given by

os_collection::maintain_order | os_collection::allow_duplicates

A run-time error is signaled if an attempt is made to create a list that is not ordered.

The **expected_size** is the cardinality you expect the collection to have when fully loaded. This value is used by ObjectStore to determine the collection's initial representation. This saves on the overhead of transforming the collection's representation as it grows during loading.

The **rep_policy** is the representation policy to be associated with the collection until explicitly changed, if **retain** is **os_collection::associate_policy**. If **retain** is **os_collection::dont_associate_policy**, the **rep_policy** is used, together with the **expected_size**, only to determine the collection's initial representation. (A representation policy is, essentially, a mapping from cardinality ranges to representation types — see **os_coll_rep_descriptor** on page 143, and in *ObjectStore Advanced C++ API User Guide* see **os_ptr_bag** and **os_packed_list**.)

```
static os_list &create(
    os_segment * seg,
    os_unsigned_int32 behavior = 0,
    os_int32 expected_size = 0,
    const os_coll_rep_descriptor *rep_policy = 0,
    os_int32 retain = dont_associate_policy
);
```

Creates a new list in the segment pointed to by **seg**. If the transient segment is specified, the list is allocated in transient memory. The rest of the arguments are just as described previously.

```
static os_list &create(
    os_object_cluster *clust,
    os_unsigned_int32 behavior = 0,
    os_int32 expected_size = 0,
    const os_coll_rep_descriptor *rep_policy = 0,
    os_int32 retain = dont_associate_policy
);
```

Creates a new list in the object cluster pointed to by **clust**. The rest of the arguments are just as described previously.

```
static os_list &create(
    void * proximity,
    os_unsigned_int32 behavior = 0,
    os_int32 expected_size = 0,
    const os_coll_rep_descriptor *rep_policy = 0,
```

```
    os_int32 retain = dont_associate_policy
);
```

Creates a new list in the segment occupied by the object pointed to by **proximity**. If the object is part of an object cluster, the new list is allocated in that cluster. If the specified object is transient, the list is allocated in transient memory. The rest of the arguments are just as described previously.

os_list::default_behavior()

static os_unsigned long default_behavior();

Returns a bit pattern indicating this type's default behavior. The default behavior is to maintain order and allow duplicates.

os_list::destroy()

static void destroy(os_list&);

Deletes the specified collection and deallocates associated storage.

Note: The assignment operator semantics are described below in terms of insert operations into the target collection. Describing the semantics in terms of insert operations serves to illustrate how duplicate, null, and order semantics are enforced. The actual implementation of the assignment might be quite different, while still maintaining the associated semantics.

os_list::operator =()

os_list &operator =(const os_collection &s);

os_list &operator = (const os_list &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_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. The result does not maintain cursors or signal duplicates.

os_collection &operator |(const void *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 or nulls, the result does. The result does not allow nulls, maintain cursors, or signal duplicates.

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_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. The result does not maintain cursors or signal duplicates.

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, the result does. If **this** allows nulls, the result does. The result does not maintain cursors or signal duplicates.

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::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 **s** is ordered, the result is. The result does not maintain cursors or signal duplicates.

os_collection &operator –(const void *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 or nulls, the result does. The result does not maintain cursors or signal duplicates.

os_list::os_list()

os_list();

Returns an empty list.

os_list(os_collection_size);

The user should pass an **os_int32** for the **os_collection_size** actual argument. Returns an empty list whose initial implementation is based on the expectation that the specified **os_int32** indicates the approximate usual cardinality of the list, once it has been loaded with elements.

os_list(const os_list&);

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

os_list(const os_collection&);

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

os_rDictionary

template <class K, class E, class R>
class os_rDictionary<K, E, R> :

Dictionaries are unordered collections that allow duplicate elements and associate 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 not be of the type **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.

Unlike the create operations for other collection classes, there are no arguments relating to representation in the **os_Dictionary** class. To control the representation for dictionaries, you use the class **os_rDictionary**, which records its elements as references. Using references can eliminate address space reservation and reduce relocation overhead.

The set of functions in the **os_rDictionary** class is identical to the set for **os_Dictionary**, with the difference that in addition to the key type and element type parameters, functions of the class **os_rDictionary** have a reference type parameter whose value must be an ObjectStore reference types.

Persistent and transient dictionaries

If you use persistent dictionaries, you must call the macro **OS_MARK_RDICTIONARY()** in your schema source file for each key-type / element-type / **os_reference** type triplet that you use. If you are using only transient dictionaries, call the macro **OS_TRANSIENT_RDICTIONARY()** in your source file.

Required header files

Programs that use the class **os_rDictionary** must include these header files: **<ostore/ostore.hh>** followed by **<ostore/coll.hh>** and **<ostore/coll/rdict_pt.hh>**. **<ostore/coll/rdict_pt.cc>** must be included in any source file that instantiates an **os_rDictionary**.

Required libraries

Programs that use the class **os_rDictionary** must link with the library file **oscol.lib** (UNIX platforms) or **oscol.ldb** (Windows platforms).

Creating **os_rDictionary** collections

Create collections with the member **create()** or, for stack-based or embedded collections, with a constructor. Do not use **new** to create collections.

Keys, elements, and references

For **os_rDictionary**, the key can be a value of any C++ fundamental type or user-defined class. When you insert an element into a dictionary, you specify the key along with the element and reference type. You can retrieve an element with a given key or retrieve those elements whose keys fall within a given range.

The element type of any instance of **os_rDictionary** must be a pointer type.

The reference type of any instance of **os_rDictionary** must be **os_reference.**

Classes used as keys

Requirements for classes used as keys are listed below.

- Types used as keys must have a no-arg constructor.
- The no-arg constructor should not allocate anything.
- Types used as keys also need a public **operator=** and copy constructor.

These requirements apply to **os_rDictionaries** that do not have the **maintain_key_order** flag set at creation time. If the **maintain_key_order** flag is on, ObjectStore does not run any user code when manipulating keys.

For class keys, the class must have a destructor.

Integer keys

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 insert. If the key type is **char*** and the dictionary has the behavior **maintain_key_order**, then it will point to the string rather than making 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.

Although it is possible to set up an **os_Cursor** on an **os_rDictionary**, you cannot iterate through it while you are doing insertions and removals from the **os_rDictionary** (safe cursor). That is, **os_rDictionary** does not support the behavior **os_collection::maintain_cursors**.

Below are two tables. The first table lists the member functions that can be performed on instances of **os_rDictionary**. The second table lists the enumerators used by **os_rDictionary.** Many functions and enumerators are inherited by **os_rDictionary** from internal collection classes. The full explanation of each inherited function or enumerator appears in the documentation for **os_collection** or **os_Collection**, as specified. The full explanation of each function and enumerator defined by **os_rDictionary** 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***
add_index	**(const os_index_path&, os_int32 = unordered, os_segment* = 0)**	**void**	**os_collection**
	(const os_index_path&, os_int32 = unordered, os_database* = 0)	**void**	
	(const os_index_path&, os_segment* = 0)	**void**	
	(const os_index_path&, os_database = 0)	**void**	

Name	*Arguments*	*Returns*	*Defined By*
cardinality	**( ) const**	**os_unsigned_int32**	**os_collection**
change_ behavior	**(os_unsigned_int32 behavior_ enums, os_int32 = verify)**	**void**	**os_collection**
clear	**( )**	**void**	**os_collection**
contains	**(const E element)**	**os_boolean**	**os_Collection**
	(const K &key_ref, const E element) const	**os_boolean**	**os_rDictionary**
	(const K *key_ptr, const E element) const		
count	**(const E) const**	**os_int32**	**os_Collection**
count_values	**(const K &key_ref) const**	**os_int32**	**os_Dictionary**
	(const K * key_ptr) const	**os_unsigned_int32**	
create (static)	**(os_segment *seg, os_unsigned_int32 expected_card = 10, os_unsigned_int32 behavior_enums = 0)**	**os_rDictionary <K,E,R>&**	**os_rDictionary**
	(os_database *db, os_unsigned_int32 expected_card = 10, os_unsigned_int32 behavior_enums = 0)	**os_Dictionary <K,E,R>&**	
	(os_object_cluster *clust, os_unsigned_int32 expected_card = 10, os_unsigned_int32 behavior_enums = 0)	**os_Dictionary <K,E,R>&**	
	(os_object_cluster *proximity, os_unsigned_int32 expected_card = 10, os_unsigned_int32 behavior_enums = 0)	**os_Dictionary <K,E,R>&**	
default_ behavior (static)	**( )**	**os_unsigned_int32**	**os_Dictionary**
destroy (static)	**(os_rDictionary<K, E.R>&)**	**void**	**os_Dictionary**
drop_index	**(const os_index_path&)**	**void**	**os_collection**
empty	**( )**	**os_int32**	**os_collection**
exists	**(char *element_type_name, char *query_string, os_database *schema_database = 0, char *file, os_unsigned_int32 line) const**	**os_int32**	**os_collection**
	(const os_bound_query&) const	**os_int32**	
get_behavior	**( ) const**	**os_unsigned_int32**	**os_collection**
has_index	**(const os_index_path&, os_int32 index_options = 0) const**	**os_int32**	**os_collection**

Name	*Arguments*	*Returns*	*Defined By*
insert	**(const K &key_ref, const E element)**	**void**	**os_rDictionary**
	(const K *key_ptr, const E element)	**void**	
only	**( ) const**	**E**	**os_Collection**
os_Dictionary	**(os_unsigned_int32 expected_card = 10, os_unsigned_int32 behavior = 0)**		**os_rDictionary**
pick	**(const os_coll_range&) const**	**E**	**os_rDictionary**
	(const K &key_ref) const	**E**	
	(const K *key_ptr) const	**E**	
	( ) const	**E**	
query	**(char *element_type_name, char *query_string, os_database *schema_database = 0, char *filename = 0, os_unsigned_int32 line = 0, os_boolean dups = query_dont_preserve_duplicates) const**	**os_Collection<E>&**	**os_Collection**
	(const os_bound_query& , os_boolean dups = query_dont_preserve_duplicates) const	**os_Collection<E>&**	
query_pick	**(char *element_type_name, char *query_string, os_database *schema_database = 0, char *filename = 0, os_unsigned_int32 line = 0) const**	**E**	**os_Collection**
	(const os_bound_query&) const	**jelm E**	
remove	**(const K &key_ref, const E element)**	**void**	**os_rDictionary**
	(const K *key_ptr, const E element)	**void**	
remove_value	**(const K &key_ref, const E os_unsigned_int32 n = 1)**	**E**	**os_rDictionary**
	(const K *key_ptr, os_unsigned_int32 n = 1)	**E**	
retrieve	**(const os_cursor&) const**	**E**	**os_rDictionary**
retrieve_key	**(const os_cursor&)**	**K***	**os_rDictionary**

os_rDictionary enumerators

The following table lists enumerators for the **os_rDictionary** class.

Name	*Inherited From*
allow_nulls	**os_collection**
associate_policy	**os_collection**
dont_associate_policy	**os_collection**
dont_verify	**os_collection**
EQ	**os_collection**
GT	**os_collection**
LT	**os_collection**
maintain_cursors	**os_collection**
maintain_key_order	**os_rDictionary**
maintain_order	**os_collection**
pick_from_empty_returns_null	**os_collection**
no_dup_keys	**os_rDictionary**
signal_cardinality	**os_collection**
signal_dup_keys	**os_rDictionary**
signal_duplicates	**os_collection**
unordered	**os_collection**
verify	**os_collection**

os_rDictionary::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_rDictionary::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_rDictionary::create()

```
static os_rDictionary<K, E, R> &create(
    os_database *db,
    os_unsigned_int32 expected_cardinality = 10,
    os_unsigned_int32 behavior_enums = 0
);
```

Creates a new dictionary in the database pointed to by **db**. If the transient database is specified, the dictionary is allocated in transient memory. **K** can be either a pointer, a basic type, or a class type. **R** is always **os_reference**.

This is one of three overloadings of **create()**. As with the create operations for the other types of collections, these overloadings differ only in the first argument, which specifies where to allocate the new dictionary. Depending on the overloading, it specifies a database, segment, or object cluster.

Usage note

For **os_rDictionary::create()**, the cardinality and behavior arguments are the third and fourth arguments to the function. This differs from **os_collection::create()**, where the behavior argument *precedes* the expected size argument.

db: The database to which the new dictionary will be allocated.

expected_cardinality: Unlike the create operations for other collection classes, there are no arguments relating to representation policies. This is because you cannot directly control the representation for dictionaries.

By default, dictionaries are presized with a representation suitable for cardinality 10. If you want a new dictionary presized for a different cardinality, supply the **expected_cardinality** argument explicitly.

If the key type is a class type, then the rank / hash functions for this type must be defined and registered through the **os_index_key()** macro.

behavior_enums: Every dictionary has the following properties:

- Its entries have no intrinsic order.
- 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:

- Performing **pick()** on an empty dictionary raises an err_coll_empty exception.
- Duplicate keys are allowed; that is, two or more elements can have the same key.
- Range lookups are not supported; that is, key order is not maintained.

You can customize the behavior of new dictionaries with regard to these last three properties. You do this by supplying a **behavior** argument to **create()**, an unsigned 32-bit integer, a bit pattern indicating the collection's properties. The bit pattern is obtained by forming the bit-wise disjunction (using bit-wise or, **|**) of enumerators taken from the following possibilities:

- **os_collection::pick_from_empty_returns_null**: Performing **pick()** on an empty dictionary returns **0** rather than raising an exception.
- **os_dictionary::signal_dup_keys**: Duplicate keys are not allowed; err_am_dup_key is signaled if an attempt is made to establish two or more elements with the same key.
- **os_dictionary::maintain_key_order**: Range lookups are supported using **pick()** or restricted cursors.

For example:

os_rDictionary<K,E,R>::default_behavior(), or

os_rDictionary<K,E,R>::create
(db,n,os_collection::pick_from_empty_returns_null)

These enumerators are instances of an enumeration defined in the scope of the **os_rDictionary**. Each enumerator is associated with a different bit, and including an enumerator in the disjunction sets its associated bit.

You can change the behavior **pick_from_empty_returns_null** after an **os_rDictionary** has been created. See **os_collection::change_behavior()** on page 97.

For large dictionaries that maintain key order, there is also an option for reducing contention. With **os_collection::dont_maintain_cardinality** behavior, **insert()** and **remove()** do not update cardinality information, avoiding contention in the collection header. This 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. See the following members of **os_collection**:

os_collection::cardinality_is_maintained() on page 97

os_collection::cardinality_estimate() on page 97

os_collection::update_cardinality() on page 128

```
static os_rDictionary<K, E, R> &create(
    os_segment *seg,
    os_unsigned_int32 expected_cardinality = 10,
    os_unsigned_int32 behavior = 0
);
```

Creates a new dictionary in the segment pointed to by **seg**. If the transient segment is specified, the dictionary is allocated in transient memory.

This is one of three overloadings of **create()**. As with the create operations for the other types of collections, these overloadings differ only in the first argument, which specifies where to allocate the new dictionary. Depending on the overloading, it specifies a database, segment, or object cluster.

The rest of the arguments are just as described previously for the first overloading of this function.

```
static os_rDictionary<K, E, R> &create(
    os_object_cluster *clust,
    os_unsigned_int32 expected_cardinality = 10,
    os_unsigned_int32 behavior = 0
```

);

Creates a new dictionary in the **os_object_cluster** pointed to by **clust**.

This is one of three overloadings of **create()**. As with the create operations for the other types of collections, these overloadings differ only in the first argument, which specifies where to allocate the new dictionary. Depending on the overloading, it specifies a database, segment, or object cluster.

The rest of the arguments are just as described previously for the first overloading of this function.

os_rDictionary::destroy()

static void destroy(os_rDictionary<K, E, R>& c);

Deletes the specified collection and deallocates associated storage.

This function has the same effect as deleting the **os_rDictionary** object.

os_rDictionary::insert()

void insert(const K &key, 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.

For **os_rDictionary <K,E,R>::insert** the element is automatically converted to an **os_reference** so that the pointer is not stored in the **os_rDictionary**.

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 insert 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 the same key as an element already present, err_am_dup_key is signaled.

void insert(const K *key, const E element)

Inserts the specified element with the key pointed to by **key_ptr**. This overloading of **insert()** differs from the preceding overloading of **insert()** only in that the key is specified with a pointer instead of a reference.

os_rDictionary::os_rDictionary()

```
os_rDictionary(
    os_unsigned_int32 expected_cardinality = 10,
    os_unsigned_int32 behavior = 0
);
```

Use the dictionary constructor only to create stack-based dictionaries, or dictionaries embedded within other objects. See **os_Dictionary::create()** for more information on creating ObjectStore dictionaries.

os_rDictionary::pick()

E pick(const os_coll_range&) const;

Returns an element of **this** that satisfies the specified **os_coll_range**. Even though the **os_rDictionary** contains elements that are stored as **os_references**, this function converts the **os_reference** element to a pointer and returns a pointer. The dictionary must be created with **maintain_key_order** to support **pick()** with **os_coll_range**.

If there is more than one such element, an arbitrary one is picked and returned. If there is no such element, **0** is returned.

E pick(const K &key_ref) const;

Returns an element of **this** that has the key referred to by **key_ref**. The value of the object referred to by **key_ref** is used for the test.

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 and has **pick_from_empty_returns_null** behavior, **0** is returned. If the dictionary is empty and does not have **pick_from_empty_returns_null** behavior, err_coll_empty is signaled.

E pick(const K *key_ptr) const;

Returns an element of **this** that has the key with the same value as ***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 and has **pick_from_empty_returns_null** behavior, **0** is returned. If the dictionary is empty and does not have **pick_from_empty_returns_null** behavior, err_coll_empty is signaled.

E pick() const;

Picks an arbitrary element of **this** and returns it.

If the dictionary is empty and has **pick_from_empty_returns_null** behavior, **0** is returned. If the dictionary is empty and does not have **pick_from_empty_returns_null** behavior, err_coll_empty is signaled.

os_rDictionary::query()

os_rDictionary::query_pick()

```
E query_pick(
    char *element_type,
    char *query_string,
    os_database *schema_db = 0,
    char *file_name = 0,
    os_unsigned_int32 line = 0
) const;
```

Returns an element (pointer) of **this** that satisfies the specified **query_string**. See the description of **query_string** for **os_Collection::query()** on page 76.

If there is no such element, **0** is returned.

Since the **os_rDictionary** stores its elements as **os_references**, doing a query will require that each element's **os_reference** be resolved. This will increase query time. Ideally there would be a reference-based index that the query can use.

The argument **element_type** is the name of the element type of **this**. Names established through the use of **typedef** are not allowed.

The **schema_db** is a database whose schema contains all the types mentioned in **query_string**. This database provides the environment in which the query is analyzed and optimized. The database in which the collection resides is often appropriate.

void *query_pick(const os_bound_query&) const;

Returns an element of **this** that satisfies the specified bound query.

If there is no such element, **0** is returned.

os_rDictionary::remove()

void remove(const K &key_ref, const E element);

Removes the dictionary entry with the element **element** at the key value 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, then 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 referred to by **key_ptr**. This overloading differs from the preceding overloading of **remove()** 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_rDictionary::remove_value()

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

Removes **n** dictionary entries with the key value 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. If removing this element leaves no other elements at this key value, the key is removed and deleted.

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 removed 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_value(const K *key_ptr, os_unsigned_int32 n = 1);

Removes **n** dictionary entries with the key pointed to by **key_ptr**. This overloading differs from the previous overloading of **insert()** 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_rDictionary::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.

os_rDictionary::retrieve_key()

const K *retrieve_key(const os_cursor&) const;

Returns the key of 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.

os_rep

An instance of this class is used to build an **os_rep_policy** or **os_rep_list**. Each **os_rep** serves to associate a collection representation type with a threshold cardinality. The threshold is the largest cardinality at which the associated representation applies.

os_rep::os_rep()

os_rep(os_rep_type rep_enum, os_unsigned_int32 threshold);

Creates an **os_rep** that specifies a representation of **rep_enum** until cardinality **threshold**. **rep_enum** should be one of the following:

- **os_packed_list_rep**
- **os_ordered_ptr_hash_rep**
- **os_ptr_bag_rep**
- **os_chained_list_rep**
- **os_ixonly_rep**
- **os_ixonly_bc_rep**
- **os_dyn_hash_rep**
- **os_dyn_bag_rep**
- **os_vdyn_hash_rep_os_reference**
- **os_vdyn_bag_rep_os_reference**

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 235). This means that when specifying **os_Set** as a function's formal parameter, or as the type of a variable or data member, you must specify the parameter (the set's *element type*). This is accomplished by appending to **os_Set** the name of the element type enclosed in angle brackets, < >:

os_Set<*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_Set** must be a pointer type.

Create collections with the member **create()** or, for stack-based or embedded collections, with a constructor. Do not use **new** to create 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 sets must include the header file **<ostore/coll.hh>** after including **<ostore/ostore.hh>**.

Required libraries

Programs that use sets must link with the library file **oscol.lib** (UNIX platforms) or **oscol.ldb** (Windows platforms).

Below are two tables. The first table 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***
add_index	**(const os_index_path&, os_int32 = unordered, os_segment* = 0)**	**void**	**os_collection**
	(const os_index_path&, os_int32 = unordered, os_database* = 0)	**void**	
	(const os_index_path&, os_segment* = 0)	**void**	
	(const os_index_path&, os_database* = 0)	**void**	
cardinality	**( ) const**	**os_int32**	**os_collection**
change_behavior	**(os_unsigned_int32 behavior, os_int32 = verify)**	**void**	**os_collection**
change_rep	**(os_unsigned_int32 expected_size, const os_coll_rep_descriptor *policy = 0, os_int32 retain = dont_associate_policy)**	**void**	**os_collection**
clear	**( )**	**void**	**os_collection**
contains	**(const E) const**	**os_int32**	**os_Collection**
count	**(const E) const**	**os_int32**	**os_Collection**
create (static)	**(os_database *db, os_unsigned_int32 behavior = 0, os_int32 expected_size = 0, const os_coll_rep_descriptor* = 0, os_int32 retain = dont_associate_policy)**	**os_Set<E>&**	**os_Set**
	(os_segment *seg, os_unsigned_int32 behavior = 0, os_int32 expected_size = 0, const os_coll_rep_descriptor* = 0, os_int32 retain = dont_associate_policy)	**os_Set<E>&**	
	(os_object_cluster *clust, os_unsigned_int32 behavior = 0, os_int32 expected_size = 0, const os_coll_rep_descriptor* = 0, os_int32 retain = dont_associate_policy)	**os_Set<E>&**	

Name	*Arguments*	*Returns*	*Defined By*
	(void* proximity, os_unsigned_int32 behavior = 0, os_int32 expected_size = 0, const os_coll_rep_descriptor* = 0, os_int32 retain = dont_associate_policy)	os_Set<E>&	
default_behavior (static)	( )	os_unsigned_int32	os_Set
destroy (static)	(os_Set<E>&)	void	os_Set
drop_index	(const os_index_path&)	void	os_collection
empty	( )	os_int32	os_collection
exists	(char *element_type_name, char *query_string, os_database *schema_database = 0, char* filename, os_unsigned_int32 line) const	os_int32	os_collection
	(const os_bound_query&) const	os_int32	
get_behavior	( ) const	os_unsigned_int32	os_collection
get_rep	( ) const	const os_coll_rep_ descriptor&	os_collection
has_index	(const os_index_path&, os_int32 index_options = unordered) const	os_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
operator const os_Bag<E>&	( ) const		os_Collection
operator os_bag&	( )		os_collection

Name	*Arguments*	*Returns*	*Defined By*
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&	**( )**		**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**	
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**	
operator =	**(const os_Set<E>&) const**	**os_Set<E>&**	**os_Set**
	(const os_Collection<E>&) const	**os_Set<E>&**	
	(E) const	**os_Set<E>&**	
operator |=	**(const os_Collection<E>&) const**	**os_Set<E>&**	**os_Set**
	(E) const	**os_Set<E>&**	
operator |	**(const os_Collection<E>&) const**	**os_Collection<E>&**	**os_Collection**
	(E) const	**os_Collection<E>&**	
operator &=	**(const os_Collection<E>&) const**	**os_Set<E>&**	**os_Set**
	(E) const	**os_Set<E>&**	

Name	*Arguments*	*Returns*	*Defined By*
operator &	**(const os_Collection<E>&) const**	**os_Collection<E>&**	**os_Collection**
	(E) const	**os_Collection<E>&**	
operator –=	**(const os_Collection<E>&) const**	**os_Set<E>&**	**os_Set**
	(E) const	**os_Set<E>&**	
operator -	**(const os_Collection<E>&) const**	**os_Collection<E>&**	**os_Collection**
	(E) const	**os_Collection<E>&**	
os_Set	**( )**		**os_Set**
	(os_collection_size)		
	(const os_Set<E>&)		
	(const os_Collection<E>&)		
pick	**( ) const**	**E**	**os_Collection**
	(const os_index_path&, const os_coll_range&) const	**E**	
query	**(char *element_type_name, char *query_string, os_database *schema_database = 0, char* file, os_unsigned_int32 line) const**	**os_Collection<E>&**	**os_Collection**
	(const os_bound_query&) const	**os_Collection<E>&**	
query_pick	**(char *element_type_name, char *query_string, os_database *schema_database = 0, char* file, os_unsigned_int32 line) const**	**E**	**os_Collection**
	(const os_bound_query&) const	**E**	
remove	**(const E)**	**os_int32**	**os_Collection**
remove_at	**(const os_Cursor<E>&)**	**void**	**os_Set**
replace_at	**(const E, const os_Cursor<E>&)**	**E**	**os_Set**
retrieve	**(const os_Cursor<E>&) const**	**E**	**os_Set**

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**

Name	***Inherited From***
associate_policy	**os_collection**
dont_associate_policy	**os_collection**
dont_verify	**os_collection**
EQ	**os_collection**
GT	**os_collection**
LT	**os_collection**
maintain_cursors	**os_collection**
maintain_order	**os_collection**
order_by_address	**os_collection**
pick_from_empty_returns_null	**os_collection**
signal_cardinality	**os_collection**
signal_duplicates	**os_collection**
unordered	**os_collection**
verify	**os_collection**

os_Set::create()

```
static os_Set<E> &create(
    os_database *db,
    os_unsigned_int32 behavior = 0,
    os_int32 expected_size = 0,
    const os_coll_rep_descriptor *rep_policy = 0,
    os_int32 retain = dont_associate_policy
);
```

Creates a new set in the database pointed to by **db**. If the transient database is specified, the set is allocated in transient memory.

The **behavior** is a bit pattern, the bit-wise disjunction (using the operator |) of enumerators indicating the desired properties. The enumerators are

- **os_collection::allow_nulls**
- **os_collection::signal_duplicates**
- **os_collection::pick_from_empty_returns_null**
- **os_collection::maintain_cursors**

See the class **os_collection** on page 87 for an explanation of each enumerator.

A run-time error is signaled if an attempt is made to create a set that is ordered or allows duplicates.

The **expected_size** is the cardinality you expect the collection to have when fully loaded. This value is used by ObjectStore to determine the collection's initial representation. This saves on the overhead of transforming the collection's representation as it grows during loading.

Representation policy

The default representation for a set is

- An **os_Set** created as an embedded object has a representation of **os_tiny_array** (0 to 4 elements).
- An embedded set becomes *out of line* and mutates to an **os_chained_list** when the fifth element is inserted.
- A set created with **::create** and a cardinality of <= 20 is represented as an **os_chained_list**.
- Once the set grows past 20 its representation is **os_dyn_hash** unless it has **maintain_cursors** behavior, in which case the representation is **os_packed_list**.

The **rep_policy** is the representation policy to be associated with the collection until explicitly changed, if **retain** is **os_collection::associate_policy**. If **retain** is **os_collection::dont_associate_policy**, the **rep_policy** is used, together with the **expected_size**, only to determine the collection's initial representation. (A representation policy is, essentially, a mapping from cardinality ranges to representation types — see **os_coll_rep_descriptor** on page 143, and in *ObjectStore Advanced C++ API User Guide* see **os_ptr_bag** and **os_packed_list**.)

An **os_Set** can have the following additional behaviors:

- **pick_from_empty_returns_null**
- **signal_duplicates**
- **allow_nulls**

```
static os_Set<E> &create(
    os_segment * seg,
    os_unsigned_int32 behavior = 0,
    os_int32 expected_size = 0,
    const os_coll_rep_descriptor *rep_policy = 0,
    os_int32 retain = dont_associate_policy
);
```

Creates a new set in the segment pointed to by **seg**. If the transient segment is specified, the set is allocated in transient memory. The rest of the arguments are just as described previously.

```
static os_Set<E> &create(
    os_object_cluster *clust,
    os_unsigned_int32 behavior = 0,
    os_int32 expected_size = 0,
    const os_coll_rep_descriptor *rep_policy = 0,
    os_int32 retain = dont_associate_policy
);
```

Creates a new set in the object cluster pointed to by **clust**. The rest of the arguments are just as described previously.

```
static os_Set<E> &create(
    void * proximity,
    os_unsigned_int32 behavior = 0,
    os_int32 expected_size = 0,
    const os_coll_rep_descriptor *rep_policy = 0,
    os_int32 retain = dont_associate_policy
);
```

Creates a new set in the segment occupied by the object pointed to by **proximity**. If the object is part of an object cluster, the new set is allocated in that cluster. If the specified object is transient, the set is allocated in transient memory. The rest of the arguments are just as described previously.

os_Set::default_behavior()

```
static os_unsigned_int32 default_behavior();
```

Returns a bit pattern indicating this type's default behavior.

os_Set::destroy()

```
static void destroy(os_Set<E>&);
```

Deletes the specified collection and deallocates associated storage. This is the same as deleting the **os_Set**.

Assignment Operator Semantics

Note: The assignment operator semantics are described below in terms of insert operations into the target collection. Describing the semantics in terms of insert operations serves to illustrate how duplicate, null, and order semantics are enforced. The actual implementation of the assignment might be quite 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_collection_size);

The user should pass an **os_int32** for the **os_collection_size** actual argument. Returns an empty set whose initial implementation is based on the expectation that the specified **os_int32** indicates the approximate usual cardinality of the set, once it 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 225.

Required header files

Programs that use sets must include the header file **<ostore/coll.hh>** after including **<ostore/ostore.hh>**.

Required libraries

Programs that use sets must link with the library file **oscol.lib** (UNIX platforms) or **oscol.ldb** (Windows platforms).

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.

Below are two tables. The first table 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***
add_index	**(const os_index_path&, os_int32 = unordered, os_segment* = 0)**	**void**	**os_collection**
	(const os_index_path&, os_int32 = unordered, os_database* = 0)	**void**	
	(const os_index_path&, os_segment* = 0)	**void**	
	(const os_index_path&, os_database* = 0)	**void**	
cardinality	**( ) const**	**os_int32**	**os_collection**

Name	*Arguments*	*Returns*	*Defined By*
change_behavior	**(os_unsigned_int32 behavior, os_int32 = verify)**	**void**	**os_collection**
change_rep	**(os_unsigned_int32 expected_size, const os_coll_rep_descriptor *policy = 0, os_int32 retain = dont_associate_policy)**	**void**	**os_collection**
clear	**( )**	**void**	**os_collection**
contains	**(const void*) const**		**os_collection**
count	**(const void*) const**	**os_int32**	
create (static)	**(os_segment *seg, os_unsigned_int32 behavior = 0, os_int32 expected_size = 0, const os_coll_rep_descriptor* = 0, os_int32 retain = dont_associate_policy)**	**os_set&**	**os_set**
	(os_database *db, os_unsigned_int32 behavior = 0, os_int32 expected_size = 0, const os_coll_rep_descriptor* = 0, os_int32 retain = dont_associate_policy)	**os_set&**	
	(os_object_cluster *clust, os_unsigned_int32 behavior = 0, os_int32 expected_size = 0, const os_coll_rep_descriptor* = 0, os_int32 retain = dont_associate_policy)	**os_set&**	
	(void* proximity, os_unsigned_int32 behavior = 0, os_int32 expected_size = 0, const os_coll_rep_descriptor* = 0, os_int32 retain = dont_associate_policy)	**os_set&**	
default_behavior (static)	**( )**	**os_unsigned_int32**	**os_set**
destroy (static)	**(os_set&)**	**void**	**os_set**
drop_index	**(const os_index_path&)**	**void**	**os_collection**
empty	**( )**	**os_int32**	**os_collection**
exists	**(char *element_type_name, char *query_string, os_database *schema_database = 0, char* file, os_unsigned_int32 line) const**	**os_int32**	**os_collection**
	(const os_bound_query&) const	**os_int32**	
get_behavior	**( ) const**	**os_unsigned_int32**	**os_collection**

Name	*Arguments*	*Returns*	*Defined By*
get_rep	**( ) const**	**os_coll_rep_ descriptor&**	**os_collection**
has_index	**(const os_index_path&, os_int32 index_options = unordered) const**	**os_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**	**os_int32**	**os_collection**
	(const void*) const	**os_int32**	
operator !=	**(const os_collection&) const**	**os_int32**	**os_collection**
	(const void*) const	**os_int32**	
operator <	**(const os_collection&) const**	**os_int32**	**os_collection**
	(const void*) const	**os_int32**	
operator <=	**(const os_collection&) const**	**os_int32**	**os_collection**
	(const void*) const	**os_int32**	
operator >	**(const os_collection&) const**	**os_int32**	**os_collection**
	(const void*) const	**os_int32**	
operator >=	**(const os_collection&) const**	**os_int32**	**os_collection**
	(const void*) const	**os_int32**	
operator =	**(const os_set&) const**	**os_set&**	**os_set**
	(const os_collection&) const	**os_set&**	
	(const void*) const	**os_set**	
operator |=	**(const os_collection&) const**	**os_set&**	**os_set**
	(const void*) const	**os_set&**	

Name	Arguments	Returns	Defined By
operator |	(const os_collection&) const	os_set&	os_set
	(const void*) const	os_set&	
operator &=	(const os_collection&) const	os_set&	os_set
	(const void*) const	os_set&	
operator &	(const os_collection&) const	os_set&	os_set
	(const void*) const	os_set&	
operator –=	(const os_collection&) const	os_set&	os_set
	(const void*) const	os_set&	
operator -	(const os_collection&) const	os_set&	os_set
	(const void*) const	os_set&	
os_set	( )		os_set
	(os_collection_size)		
	(const os_set&)		
	(const os_collection&)		
pick	( ) const	void*	os_collection
	(const os_index_path&, const os_coll_range&) const	void*	
query	(char *element_type_name, char *query_string, os_database *schema_database = 0, char* file, os_unsigned_int32 line) const	os_collection&	os_collection
	(const os_bound_query&) const	os_collection&	
query_pick	(char *element_type_name, char *query_string, os_database *schema_database = 0, char* file, os_unsigned_int32 line) const	void*	os_collection
	(const os_bound_query&) const	void*	
remove	(const void*)	os_int32	os_collection
remove_at	(const os_cursor&)	void	os_set
replace_at	(const void*, const os_cursor&)	void*	os_set
retrieve	(const os_cursor&) const	void*	os_set

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**
associate_policy	**os_collection**
dont_associate_policy	**os_collection**
dont_verify	**os_collection**
EQ	**os_collection**
GT	**os_collection**
LT	**os_collection**
maintain_cursors	**os_collection**
maintain_order	**os_collection**
order_by_address	**os_collection**
pick_from_empty_returns_null	**os_collection**
signal_cardinality	**os_collection**
signal_duplicates	**os_collection**
unordered	**os_collection**
verify	**os_collection**

os_set::create()

```
static os_set &create(
    os_database *db,
    os_unsigned_int32 behavior = 0,
    os_int32 expected_size = 0,
    const os_coll_rep_descriptor *rep_policy = 0,
    os_int32 retain = dont_associate_policy
);
```

Creates a new set in the database pointed to by **db**. If the transient database is specified, the set is allocated in transient memory.

The **behavior** is a bit pattern, the bit-wise disjunction (using the operator |) of enumerators indicating the desired properties. The enumerators are

- **os_collection::allow_nulls**
- **os_collection::signal_duplicates**

- **os_collection::pick_from_empty_returns_null**
- **os_collection::maintain_cursors**

See the class **os_collection** on page 87 for an explanation of each enumerator.

A run-time error is signaled if an attempt is made to create a set that is ordered or allows duplicates.

The **expected_size** is the cardinality you expect the collection to have when fully loaded. This value is used by ObjectStore to determine the collection's initial representation. This saves on the overhead of transforming the collection's representation as it grows during loading.

The **rep_policy** is the representation policy to be associated with the collection until explicitly changed, if **retain** is **os_collection::associate_policy**. If **retain** is **os_collection::dont_associate_policy**, the **rep_policy** is used, together with the **expected_size**, only to determine the collection's initial representation. (A representation policy is, essentially, a mapping from cardinality ranges to representation types — see **os_coll_rep_descriptor** on page 143, and in *ObjectStore Advanced C++ API User Guide* see **os_ptr_bag** and **os_packed_list**.)

```
static os_set &create(
    os_segment * seg,
    os_unsigned_int32 behavior = 0,
    os_int32 expected_size = 0,
    const os_coll_rep_descriptor *rep_policy = 0,
    os_int32 retain = dont_associate_policy
);
```

Creates a new set in the segment pointed to by **seg**. If the transient segment is specified, the set is allocated in transient memory. The rest of the arguments are just as described previously.

```
static os_set &create(
    os_object_cluster *clust,
    os_unsigned_int32 behavior = 0,
    os_int32 expected_size = 0,
    const os_coll_rep_descriptor *rep_policy = 0,
    os_int32 retain = dont_associate_policy
);
```

Creates a new set in the object cluster pointed to by **clust**. The rest of the arguments are just as described previously.

```
static os_set &create(
    os_object_cluster *proximity,
    os_unsigned_int32 behavior = 0,
    os_int32 expected_size = 0,
    const os_coll_rep_descriptor *rep_policy = 0,
    os_int32 retain = dont_associate_policy
);
```

Creates a new set in the specified object cluster. The rest of the arguments are just as described previously.

os_set::default_behavior()

static os_unsigned_long default_behavior();

Returns a bit pattern indicating this type's default behavior.

os_set::destroy()

static void destroy(os_set&);

Deletes the specified collection and deallocates associated storage.

Note: The assignment operator semantics are described below in terms of insert operations into the target collection. Describing the semantics in terms of insert operations serves to illustrate how duplicate, null, and order behavior are enforced. The actual implementation of the assignment might be quite different, while still maintaining the associated behavior.

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_collection &s) const;

Copies the contents of **this** into a new collection, then inserts the elements of **s** into the new collection. The new collection is then returned. If **s** allows duplicates, the result does. If either operand allows nulls, the result does. The result does not maintain order, maintain cursors, or signal duplicates.

os_set &operator |(const void *e) const;

Copies the contents of **this** into a new collection, then inserts **e** into the new collection. The new collection is then returned. If **this** allows nulls, the result does. The result does not allow duplicates, maintain order, maintain cursors, or signal duplicates.

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_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 **s** allows duplicates, the result does. If either operand allows nulls, the result does. The result does not maintain order, maintain cursors, or signal duplicates.

os_set &operator &(const void *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 nulls, the result does. The result does not allow duplicates, maintain order, maintain cursors, or signal duplicates.

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::operator –()

os_set &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 **s** allows duplicates, the result does. If either operand allows nulls, the result does. The result does not maintain order, maintain cursors, or signal duplicates.

os_set &operator –(const void *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 nulls, the result does. The result does not allow duplicates, maintain order, maintain cursors, or signal duplicates.

os_set::os_set()

os_set();

Returns an empty set.

os_set(os_collection_size);

The user should pass an **os_int32** for the **os_collection_size** actual argument. Returns an empty set whose initial implementation is based on the expectation that the specified **os_int32** indicates the approximate usual cardinality of the set, once it 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.

os_set::retrieve()

void* retrieve(const os_cursor&) const;

Returns the element at which the specified cursor is positioned. 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.

Chapter 3
Representation Types

Types

os_chained_list

The class **os_chained_list** is a representation type that is optimized (in both time and space) for small- to medium-sized collections. Each **os_chained_list** consists of a header and any number of blocks. The header has a **vptr**, one word of state and up to 15 pointers. When the number of pointers in the header is exhausted, an **os_chained_list_block** is allocated and chained to the header.

Each **os_chained_list_block** can contain up to 255 pointers. It has two or three words of overhead: one word of state information, a *previous* pointer, and possibly a *next* pointer (the first **os_chained_list_block** allocated does not have a *next* pointer until the next block is allocated). The default version of **os_chained_list** contains four pointers in the header and seven or eight pointers in its blocks.

The maximum cardinality for **os_chained_list**s is 131070.

Controlling the number of pointers

When you create an **os_chained_list**, what is really allocated is an instance of a parameterized class derived from **os_chained_list**: **os_chained_list_pt<NUM_PTRS_IN_HEAD,NUM_PTRS_IN_BLOCKS>**. The default parameterization is <4,8>, but you can specify a different parameterization with the following macros:

Macros for specifying parameterization

OS_MARK_CHAINED_LIST_REP (ptrs_in_header,ptrs_in_blocks)	Use **OS_MARK_CHAINED_LIST_REP()** in the same dummy function as **OS_MARK_SCHEMA_TYPE()**.
OS_INSTANTIATE_CHAINED_LIST_REP (ptrs_in_header,ptrs_in_blocks)	Use **OS_INSTANTIATE_CHAINED_LIST_REP()** at file scope. It declares some static state needed by the representation.
OS_INITIALIZE_CHAINED_LIST_REP (ptrs_in_header,ptrs_in_blocks)	Execute **OS_INITIALIZE_CHAINED_LIST_REP()** in a function. It registers the new parameterization with the collections library.

Include the files **<coll/chlist.hh>**, **<coll/chlistpt.hh>**, and **<coll/chlistpt.c>** if you use these macros.

In order to create a collection using a chained list with other than the default parameterization, you invoke the following static member function:

```
static os_chained_list_descriptor*
os_chained_list_descriptor::find_rep(
    os_unsigned_int32 ptrs_in_hdr,
    os_unsigned_int32 ptrs_in_blocks
);
```

If the requested parameterization has been specified with the above macros, the appropriate representation descriptor is returned. Otherwise, **0** is returned.

Note that an **os_chained_list** must have at least four pointers in the header but not more than 15 pointers.

An **os_chained_list** with a four-pointer header can change freely into any other collection representation and the reverse. However, other collection representations cannot change into **os_chained_lists** with more than four pointers in the header. A normal collection header is 24 bytes. An **os_chained_list** with more than four pointers exceeds this limit. It is possible for an **os_chained_list** with an oversized header to change into another representation (with the same or smaller size header).

Pool allocation of blocks

You can request pool allocation of **os_chained_list_blocks** with the environment variable **OS_COLL_POOL_ALLOC_CHLIST_BLOCKS** and the function **os_chlist_pool::configure_pool()**. In some cases this decreases the time needed for individual allocation of **os_chained_list_blocks** and increases the chance of getting good locality of reference.

Setting **OS_COLL_POOL_ALLOC_CHLIST_BLOCKS** (to 1) turns on pool allocation. There is one pool per segment; each pool consists of an array of subpools. Each subpool is two pages by default.

By allocating larger subpools, you can defer the cost of allocating new subpools at the expense of potentially wasted space. To allocate larger subpools, use this function:

```
static void
os_chlist_pool::configure_pool(
    os_unsigned_int32 config_options,
```

```
    os_unsigned_int32 blks_per_subpool=2
);
```

config_options can have one of the following values:

- **os_chlist_pool_no_pooled_allocation**
- **os_chlist_pool_allocate_blks**

The second argument, which is optional and defaults to **2**, controls the number of pages allocated per subpool.

Mutation checks

In order to improve performance, an **os_chained_list** does not necessarily check to see if it should change to another representation after every insert or remove operation. By default, it checks when the cardinality is roughly a multiple of 7. However, you can control the frequency with which it checks by invoking the static member function

```
static void
os_chained_list_descriptor::set_reorg_check_interval(
        os_unsigned_int32 v
);
```

ObjectStore sets the check interval to one less than the power of 2 that is greater than or equal to **v**. For example, in order to check on every other insert or remove, pass 1 or 2 as an argument. Passing 3 or 4 results in a check on every third operation. Passing 0 inhibits mutation. However, if the maximum cardinality for an **os_chained_list** is reached, it will change to another representation.

mutate_when_full behavior

For collections whose representation is **os_chained_list**, if you specify the behavior enumerator **os_collection::chained_list_mutate_when_full**, the collection's representation will not change until it reaches the maximum cardinality for chained lists.

os_dyn_bag

Instances of this class are used as ObjectStore collection representations. The **os_dyn_bag** representation supports **O(1)** element lookup, which means that operations such as **contains()** and **remove()** are **O(1)** (in the number of elements). But an **os_dyn_bag** takes up somewhat more space than an **os_packed_list**.

The representation **os_dyn_bag** minimizes reorganization overhead at the expense of some extra space overhead, compared with **os_ptr_bag**. At large cardinalities, **os_dyn_bag** uses a directory structure pointing to many small hash tables that can reorganize independently.

This representation type does not support **maintain_order** or **maintain_cursors** behavior.

For cardinalities below 30, **os_chained_list** might be a better representation type.

In the following table, complexities are shown in terms of collection cardinality, represented by **n**. (These complexities reflect the nature of the computational overhead involved, not overhead due to disk I/O and network traffic.)

insert()	**O(1)**
remove()	**O(1)**
cardinality()	**O(1)**
contains()	**O(1)**
comparisons (**<=, ==,** and so on)	**O(n)**
merges (**|**, **&**, **-**)	**O(n)**

If **cardinality <= 64k,** the small-medium cardinality data structure is used. It contains the following:

- A header (24 bytes)
- An entry for each element (eight bytes each)
- Some number of empty entries (eight bytes each)

On average, an **os_dyn_bag** at low-medium cardinalities is 69% full. You can estimate the average size as follows:

Avg. total size in bytes = 24 + (cardinality/.69) * 8

If **cardinality > 64k** the large cardinality data structure is used. It contains the following:

- A header (24 bytes)
- A directory (60-byte header + 12 bytes per directory entry)
- Some number of small hash tables (two pages each, eight bytes per entry)

On average, each small hash table in an **os_dyn_bag** at high cardinalities is 70% full. You can estimate the average size as follows:

n_entries = Avg. number of entries per small hash table = (8192/8) * .7

n_tables = Avg. number of small hash tables = cardinality / n_entries

dir_size = Avg. directory size in bytes = 60 + (n_tables+1) * 12

Avg. total size in bytes = 24 bytes + dir_size + n_tables * 8192

os_dyn_hash

Instances of this class are used as ObjectStore collection representations. The dynamic hash representation supports **O(1)** element lookup, which means that operations such as **contains()** and **remove()** are **O(1)** (in the number of elements). But an **os_dyn_hash** takes up somewhat more space than an **os_packed_list**.

At large cardinalities, **os_dyn_hash** uses a directory structure pointing to many small hash tables that can reorganize independently.

This representation type does not support **allow_duplicates**, **maintain_order**, or **maintain_cursors** behavior.

For cardinalities below 30, **os_chained_list** might be a better representation type.

In the following table, complexities are shown in terms of collection cardinality, represented by **n**. (These complexities reflect the nature of the computational overhead involved, not overhead due to disk I/O and network traffic.)

insert()	**O(1)**
remove()	**O(1)**
cardinality()	**O(1)**
contains()	**O(1)**
comparisons (**<=, ==,** and so on)	**O(n)**
merges (**|**, **&**, **-**)	**O(n)**

If **cardinality <= 64k,** the small-medium cardinality data structure is used. It contains the following:

- A header (24 bytes)
- An entry for each element (four bytes each)
- Some number of empty entries (four bytes each)

On average, an **os_dyn_hash** at low-medium cardinalities is 69% full. You can estimate the average size as follows:

Avg. total size in bytes = 24 + (cardinality/.69) * 4

If **cardinality > 64k** the large cardinality data structure is used. It contains the following:

- A header (24 bytes)
- A directory (60-byte header + 12 bytes per directory entry)
- Some number of small hash tables (two pages each, four bytes per entry)

On average, each small hash table in an **os_dyn_hash** at high cardinalities is 70% full. You can estimate the average size as follows:

n_entries = Avg. number of entries per small hash table = (8192/4) * .7

n_tables = Avg. number of small hash tables = cardinality / n_entries

dir_size = Avg. directory size in bytes = 60 + (n_tables+1) * 12

Avg. total size in bytes = 24 bytes + dir_size + n_tables * 8192

os_ixonly and os_ixonly_bc

Instances of these classes are used as ObjectStore collection representations. They are both index-only representations that support **O(1)** element lookup. Operations such as **contains()** and **remove()** are **O(1)** (in the number of elements). But they take up somewhat more space than an **os_packed_list**.

For large collections subject to contention, **os_ixonly_bc** can provide significantly better performance than **os_ixonly**. See **os_ixonly_bc**, below.

The next chapter discusses associating indexes with collections to improve the efficiency of queries. With **os_ixonly** or **os_ixonly_bc**, you can save space by telling ObjectStore to record the membership of the collection in one of its indexes, as opposed to recording the membership in both the index and the collection. In other words, you can save space by using an index as a collection's representation.

When these representation types are specified for a collection, you must add an index to it before any operations are performed on it. Additional indexes can also be added.

These representation types are incompatible with the following behaviors: **maintain_order**, **maintain_cursors**, **allow_nulls**, and **allow_duplicates**.

Note that using these representations can save on space overhead at the expense of reducing the efficiency of some collection operations. If the only time-critical collection operation is index-based element lookup, an index-only representation is likely to be beneficial.

For cardinalities below 30, **os_chained_list** might be a better representation type.

os_ixonly_bc is just like **os_ixonly**, except that **insert()** and **remove()** do not update cardinality information, avoiding contention in the collection header. The disadvantage of **os_ixonly_bc** is that **cardinality()** is an **O(n)** operation, requiring a scan of the whole collection.

You can determine if a collection updates its cardinality in this way with the following member of **os_collection**:

os_int32 cardinality_is_maintained() const;

This function returns nonzero if the collection maintains cardinality; it returns **0** otherwise.

The following member of **os_collection**, which returns an estimate of a collection's cardinality, is an **O(1)** operation in the size of the collection:

os_unsigned_int32 cardinality_estimate() const;

This function returns the cardinality as of the last call to **os_collection::update_cardinality()** — see below. For collections that maintain cardinality, the actual cardinality is returned.

Before you add a new index to an **os_ixonly_bc** collection, call the following member of **os_collection**:

os_unsigned_int32 update_cardinality();

If you do not, **add_index()** will work correctly, but less efficiently than if you do. This function updates the value returned by **os_collection::cardinality_estimate()**, by scanning the collection and computing the actual cardinality.

In the following table, complexities are shown in terms of collection cardinality, represented by **n**. (These complexities reflect the nature of the computational overhead involved, not overhead due to disk I/O and network traffic.)

insert()	**O(1)**
remove()	**O(1)**
cardinality(), os_ixonly	**O(1)**
cardinality(), os_ixonly_bc	**O(n)**
contains()	**O(1)**
comparisons (**<=, ==,** and so on)	**O(n)**
merges (**|**, **&**, **-**)	**O(n)**

If there are safe cursors open on a particular collection, each insert or remove operation visits each of those cursors and adjusts them if necessary.

os_ordered_ptr_hash

Instances of this class are used as ObjectStore collection representations. Unlike the other hash tables, this representation supports **maintain_order** behavior. The ordered pointer hash representation supports **O(1)** element lookup, which means that operations such as **contains()** and **remove()** are **O(1)** (in the number of elements). But an **os_ordered_ptr_hash** takes up somewhat more space than an **os_packed_list**.

This representation type does not support **be_an_array** behavior.

For cardinalities below 30, **os_chained_list** might be a better representation type.

Time Complexity

In the following table, complexities are shown in terms of collection cardinality, represented by **n**. (These complexities reflect the nature of the computational overhead involved, not overhead due to disk I/O and network traffic.)

insert()	**O(1)**
position-based insert	**O(n)**
remove()	**O(1)**
position-based remove	**O(n)**
cardinality()	**O(1)**
contains()	**O(1)**
comparisons (**<=, ==,** and so on)	**O(n)**
merges (**|**, **&**, **-**)	**O(n)**

If there are safe cursors open on a particular collection, each insert or remove operation visits each of those cursors and adjusts them if necessary.

Space Overhead and Clustering

An ordered pointer hash has the following components:

- Header
- Entry for each element
- Some number of empty entries

The entry for a given element is likely to be on a different page from the collection header.

On average, a pointer hash is 58.3% full. You can estimate the average size of a pointer hash as follows:

if cardinality <= 65535

average total size in bytes = 56 + cardinality * 8 / 58.3

if cardinality > 65535

average total size in bytes = 56 + cardinality * 12 / 58.3

The minimum fill for a packed list is 46.7%, so an upper bound on collection space overhead can be calculated as follows:

if cardinality <= 65535

maximum total size in bytes = 56 + cardinality * 8 / 46.7

if cardinality > 65535

maximum total size in bytes = 56 + cardinality * 12 / 46.7

os_packed_list

Instances of this class are used as ObjectStore collection representations. The packed list representation is relatively space-efficient, but element lookup is an **O(n)** operation, which means that operations such as **remove()** and **contains()** are **O(n)** (in the number of elements). If duplicates are allowed, this representation provides the fastest insertion times, but if duplicates are not allowed (requiring element lookup to check for the presence of a duplicate), **insert()** is **O(n)**.

For cardinalities below 30, **os_chained_list** might be a better representation type.

In the following table, complexities are shown in terms of collection cardinality, represented by **n**. (These complexities reflect the nature of the computational overhead involved, not overhead due to disk I/O and network traffic.)

insert(), duplicates allowed	**O(1)**
insert(), duplicates not allowed	**O(n)**
position-based insert, no "holes"	**O(1)**
position-based insert, with "holes"	**O(n)**
remove()	**O(n)**
position-based remove, no "holes"	**O(1)**
position-based remove, with "holes"	**O(n)**
cardinality()	**O(1)**
contains()	**O(n)**
comparisons (**<=, ==,** and so on)	**O(n2)**
merges (**|**, **&**, **-**)	**O(n2)**

There might be “holes” in an **os_packed_list** if any elements have been removed.

If there are safe cursors open on a particular collection, each insert or remove operation visits each of those cursors and adjusts them if necessary.

A packed list has the following components:

- Header
- Entry for each element

- Some number of empty entries

The entry for a given element is likely to be on a different page from the collection header.

On average, a packed list is 83.3% full. You can estimate the average size of a collection as follows:

average total size in bytes = 40 + cardinality * 4 / 83.3

The minimum fill for a packed list is 66.7%, so an upper bound on collection space overhead can be calculated as follows:

maximum total size in bytes = 40 + cardinality * 4 / 66.7

os_ptr_bag

Instances of this class are used as ObjectStore collection representations. The pointer hash representation supports **O(1)** element lookup, which means that operations such as **contains()** and **remove()** are **O(1)** (in the number of elements). But an **os_ptr_bag** takes up somewhat more space than an **os_packed_list**.

In addition, as an **os_ptr_bag** grows, there can be overhead during collection updates, for reorganization. The representation **os_dyn_bag** minimizes reorganization overhead at the expense of some extra space overhead by using, at large cardinalities, a directory structure that points to many small hash tables that can reorganize independently.

This representation type does not support **maintain_order** behavior.

For cardinalities below 30, **os_chained_list** might be a better representation type.

Time Complexity

In the following table, complexities are shown in terms of collection cardinality, represented by **n**. (These complexities reflect the nature of the computational overhead involved, not overhead due to disk I/O and network traffic.)

insert()	**O(1)**
remove()	**O(1)**
cardinality()	**O(1)**
contains()	**O(1)**
comparisons (**<=, ==,** and so on)	**O(n)**
merges (**|**, **&**, **-**)	**O(n)**

If there are safe cursors open on a particular collection, each insert or remove operation visits each of those cursors and adjusts them if necessary.

Space Overhead and Clustering

A pointer hash has the following components:

- Header

- Entry for each element
- Some number of empty entries
- Count slot for each entry
- Some number of empty count slots

The entry for a given element is likely to be on a different page from the collection header. In addition, the count slot for a given element is likely to be stored on a different page from both the header and the entry for the element.

On average, a pointer bag is 58.3% full. You can estimate the average size of a pointer bag as follows:

average total size in bytes = 48 + cardinality * 8 / 58.3

The minimum fill for a packed list is 46.7%, so an upper bound on collection space overhead can be calculated as follows:

maximum total size in bytes = 48 + cardinality * 8 / 46.7

os_vdyn_bag

Instances of this class are used as ObjectStore collection representations. The **os_vdyn_bag** representation saves on relocation overhead by recording its membership using ObjectStore references instead of pointers. It supports **O(1)** element lookup, which means that operations such as **contains()** and **remove()** are **O(1)** (in the number of elements). But an **os_vdyn_bag** takes up more space than an **os_packed_list**.

The representation **os_vdyn_bag** minimizes reorganization overhead at the expense of some extra space overhead, compared with **os_ptr_bag**. At large cardinalities, **os_vdyn_bag** uses a directory structure pointing to many small hash tables that can reorganize independently.

This representation type does not support **maintain_order** or **maintain_cursors** behavior.

For cardinalities below 30, **os_chained_list** might be a better representation type.

This class is parameterized, with a parameter indicating the type of ObjectStore reference to use for recording membership. The parameter must be **os_reference**.

In the following table, complexities are shown in terms of collection cardinality, represented by **n**. (These complexities reflect the nature of the computational overhead involved, not overhead due to disk I/O and network traffic.)

insert()	**O(1)**
remove()	**O(1)**
cardinality()	**O(1)**
contains()	**O(1)**
comparisons (**<=, ==,** and so on)	**O(n)**
merges (**|**, **&**, **-**)	**O(n)**

For an **os_vdyn_bag** whose reference type parameter is **REF_TYPE**, if

cardinality <= 64k

the small-medium cardinality data structure is used. You can estimate its size as follows:

```
average total size = 24 bytes (header) +
    ( ((cardinality / .69) / 16)  + ((cardinality / .69) % 16)) *
    ( ((sizeof(REF_TYPE) + 4) * 16) + 4)
```

If

```
    cardinality > 64k
```

the large cardinality data structure is used. You can estimate its size as follows:

```
entry_size:

    os_reference: 20

n_tables = (cardinality / ( ((8192 / <entry-size> )*2) * .7))

dir_size= (n_tables +1) * 12 bytes + 60

average total size = 24 bytes (header) + dir_size + n_tables * 8192
bytes
```

os_vdyn_hash

Instances of this class are used as ObjectStore collection representations. The **os_vdyn_hash** representation saves on relocation overhead by recording its membership using ObjectStore references instead of pointers. It supports **O(1)** element lookup, which means that operations such as **contains()** and **remove()** are **O(1)** (in the number of elements). But an **os_vdyn_hash** takes up more space than an **os_packed_list**.

At large cardinalities, **os_vdyn_hash** uses a directory structure pointing to many small hash tables that can reorganize independently.

This representation type does not support **allow_duplicates, maintain_order**, or **maintain_cursors** behavior.

For cardinalities below 30, **os_chained_list** might be a better representation type.

This class is parameterized, with a parameter indicating the type of ObjectStore reference to use for recording membership. The parameter must be **os_reference**.

In the following table, complexities are shown in terms of collection cardinality, represented by **n**. (These complexities reflect the nature of the computational overhead involved, not overhead due to disk I/O and network traffic.)

insert()	**O(1)**
remove()	**O(1)**
cardinality()	**O(1)**
contains()	**O(1)**
comparisons (**<=, ==,** and so on)	**O(n)**
merges (**|**, **&**, **-**)	**O(n)**

For an **os_vdyn_hash** whose reference type parameter is **REF_TYPE**, if

cardinality <= 64k

the small-medium cardinality data structure is used. You can estimate its size as follows:

**average total size = 24 bytes (header) +
(((cardinality / .69) / 16) + ((cardinality / .69) % 16)) *
(sizeof(REF_TYPE) + 4)**

If

cardinality > 64k

the large cardinality data structure is used. You can estimate its size as follows:

entry_size:

os_reference: 20

n_tables = (cardinality / (((8192 / <entry-size>)) * .7))

dir_size= (n_tables +1) * 12 bytes + 60

average total size = 24 bytes (header) + dir_size + n_tables * 8192 bytes

Chapter 4
Macros and User-Defined Functions

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_QUERY_FUNCTION()

Applications that use a member function in a query or path string must call this macro.

Form of the call

OS_MARK_QUERY_FUNCTION(*class*,*func*)

class is the name of the class that defines the member function.

func is the name of the member function itself.

The **OS_MARK_QUERY_FUNCTION()** macro should be invoked along with the **OS_MARK_SCHEMA_TYPE()** macros for an application's schema, that is, in the schema source file. No white space should appear in the argument list of **OS_MARK_QUERY_FUNCTION()**.

OS_MARK_RDICTIONARY()

If you use reference-based persistent dictionaries, or any combination of persistent and transient dictionaries, you must call the macro **OS_MARK_RDICTIONARY()** for each key-type / element-type / reference-type triplet that you use.

Form of the call

OS_MARK_RDICTIONARY(*key_type*, *element_type*, *reference_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/rdict_pt.hh>
#include <ostore/manschem.hh>
#include "dnary.hh"

   OS_MARK_RDICTIONARY(void*,Course*,os_reference);
   OS_MARK_RDICTIONARY(int,Employee**,os_reference);
   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.

For *reference_type*, specify **os_reference**.

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(*element_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 the *key_type*.

For class keys, the class must have a destructor.

If the user-defined class being used as a key does not have **get_os_typespec()** declared, then the internal function **os_dk_wrapper<KEYTYPE>::_type()** (defined in **dkey.hh**) will complain about **KEYTYPE::get_os_typespec()**'s not being declared. If **get_os_typespec()** is declared but undefined, an unresolved reference link error will occur. Therefore, **get_os_typespec()** should be defined as the following, where **KEYTYPE** is the name of the user-defined class:

```
{ return new os_typespec("KEYTYPE") ; }
```

OS_TRANSIENT_RDICTIONARY()

If you use reference-based transient dictionaries (the **os_rDdictionaries** class), you must call the macro **OS_TRANSIENT_RDICTIONARY()** for each key-type / element-type / reference-type triplet that you use.

Form of the call

OS_TRANSIENT_RDICTIONARY(*key_type***,** *element_type***,** *reference_type***)**

Here are some examples:

```
OS_TRANSIENT_RDICTIONARY(void*,Course*,os_reference);
OS_TRANSIENT_RDICTIONARY(int,Employee**,os_reference);
```

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 a destructor.

os_index()

This macro is used to designate a class's **os_backptr**-valued member when calling **make_link()** and **break_link()** or when defining indexable members. The **os_backptr** member is used to establish other members of the class as indexable. Bit-field members cannot be indexable.

To use ObjectStore's collection facility, you must include the file **<ostore/coll.hh>** after including **<ostore/ostore.hh>**.

Form of the call

os_index(*class*,*member*)

class is the class defining the **os_backptr** data member.

member is the name of the **os_backptr** member.

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()

This macro is used to register user-defined rank and hash functions with ObjectStore.

To use ObjectStore's collection facility, you must include the file **<ostore/coll.hh>** after including **<ostore/ostore.hh>**.

Form of the call

os_index_key(*class*,*rank_function*,*hash_function*)

This macro must be within the scope of any query or cursor that might need the rank or hash functions.

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**. In *ObjectStore Advanced C++ API User Guide* see Rank and Hash Function Requirements on page 161.

hash_function is a user-defined 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. Supplying this function is optional. If you do not supplying 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 ObjectStore's collection facility, you must include the file **<ostore/coll.hh>** after including **<ostore/ostore.hh>**.

Form of the call

os_index_key_hash_function(*class***,***hash_function***)**

This macro must be within the scope of any query or cursor that might need the rank or hash functions.

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 a value, an **os_unsigned_int32**, 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 ObjectStore's collection facility, you must include the file **<ostore/coll.hh>** after including **<ostore/ostore.hh>**.

Form of the call

os_index_key_rank_function(*class*,*rank_function*)

This macro must be within the scope of any query or cursor that might need the rank or hash functions.

class is the class whose instances are ranked by the specified function. *class* can also be **char*** when registering **os_strcoll_for_char_pointer()**, and **char[]** when registering **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**. In *ObjectStore Advanced C++ API User Guide* see Rank and Hash Function Requirements.

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_body()

This macro is used to instantiate accessor functions for an indexable data member. Calls to this macro should appear at top level in the source file associated with the class defining the member.

To use this macro, you must include the file **<ostore/relat.hh>** after including **<ostore/ostore.hh>**.

The actual value type of an indexable data member is a special class whose instances encapsulate the member's apparent value. This implicitly defined class defines **operator =()** (for setting the apparent value) 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 indexable member's apparent value explicitly.

Form of the call

os_indexable_body(*class***,***member***,***value_type***,***index***)**

class is the class defining the data member being declared.

member is the name of the member being declared.

value_type is the (apparent) value type of the indexable member.

index is a call to the macro **os_index()**, indicating the name of the defining class's **os_backptr** member.

Caution

The first three macro arguments are used (among other things) to concatenate unique names for the encapsulating 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_indexable_member()

This macro is used to establish a data member as indexable in order to perform automatic index maintenance. Field members cannot be indexable.

To use ObjectStore's collection facility, you must include the file **<ostore/coll.hh>** after including **<ostore/ostore.hh>**.

The macro call is used instead of the value type in the member declaration.

class *class-name*
{
    ...
    *macro-call member-name***;**
    ...
};

The actual value type of an indexable data member is a special class whose instances encapsulate the member's apparent value. This implicitly defined class defines **operator =()** (for setting the apparent value) 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 indexable member's apparent value explicitly.

Form of the call

os_indexable_member(*class***,***member***,***value_type***)**

class is the class defining the data member being declared.

member is the name of the member being declared.

value_type is the (apparent) value type of the member being declared.

Caution

The first two macro arguments are used (among other things) to concatenate unique names for the encapsulating 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_query_function()

Applications that use a member function (which does or does not return a reference) in a query or path string must call this macro.

Form of the call

os_query_function(*class***,***func***,***return_type***)**

class is the name of the class defining the member function.

func is the name of the member function itself.

return_type names the type of value returned by the member function.

The **os_query_function()** macro should be invoked at module level in a header file (for example, the file containing the definition of the class that declares the member function). No white space should appear in the argument list.

os_query_function_body()

Applications that use a member function in a query or path string must call this macro.

Form of the call

os_query_function_body(*class***,***func***,***return_type***,***bpname***)**

class is the name of the class that defines the member function.

func is the name of the member function itself.

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()** 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_query_function_body_returning_ref()

This macro enables users to register a query function that returns a reference. The application that uses this member function in a query must call **os_query_function_body_returning_ref()**.

Form of the call

os_query_function_body_returning_ref(*class***,***func***,***return_type***,***bpname***)**

where

- *class* is the name of the class defining the member function.
- *func* is the name of the member function itself.
- *return_type* names the type of value returned by the member function. The way to use this is to pass just *return_type*, not *return_type***&**, to the *return_type* arguments of the macro.
- *bpname* is the name of the **os_backptr-**valued member of **class**.

os_query_function_returning_ref()

The application that uses this member function, returning a reference, in a query must call **os_query_function_returning_ref()**. A call to this macro has the form

Form of the call

os_query_function_returning_ref(*class***,***func***,***return_type***)**

where

- *class* is the name of the class defining the member function
- *func* is the name of the member function itself.
- *return_type* names the type of value returned by the member function. The way to use this is to pass just *return_type*, not *return_type***&**, to the macro *return_type* arguments.

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, 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.

Required include files

To use this macro, you must include the file **<ostore/relat.hh>** after including **<ostore/ostore.hh>**. If you also include **<ostore/coll.hh>**, include **<ostore/relat.hh>** after both **<ostore/ostore.hh>** and **<ostore/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 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.

Required include files

To use this macro, you must include the file **<ostore/relat.hh>** after including **<ostore/ostore.hh>** and **<ostore/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 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.

Required include files

To use this macro, you must include the file **<ostore/relat.hh>** after including **<ostore/ostore.hh>** and **<ostore/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 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.

Required include files

To use this macro, you must include the file **<ostore/relat.hh>** after including **<ostore/ostore.hh>** and **<ostore/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 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.

Required include files

To use this macro, you must include the file **<ostore/relat.hh>** after including **<ostore/ostore.hh>**. If you also include **<ostore/coll.hh>**, include **<ostore/relat.hh>** after both **<ostore/ostore.hh>** and **<ostore/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 or when either member is indexable. Calls to this macro should appear at 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**. For indexable members, use a call to the macro **os_index()**, indicating the name of the defining class's **os_backptr** member.

inv_index specifies whether the inverse member is indexable. For nonindexable members, use **os_no_index**. For indexable members, use a call to the macro **os_index()**, indicating the name of the defining class's **os_backptr** member.

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.

Required include files

To use this macro, you must include the file **<ostore/relat.hh>** after including **<ostore/ostore.hh>** and **<ostore/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 or when either member is indexable. Calls to this macro should appear at 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**. For indexable members, use a call to the macro **os_index()**, indicating the name of the defining class's **os_backptr** member.

inv_index specifies whether the inverse member is indexable. For nonindexable members, use **os_no_index**. For indexable members, use a call to the macro **os_index()**, indicating the name of the defining class's **os_backptr** member.

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.

Required include files

To use this macro, you must include the file **<ostore/relat.hh>** after including **<ostore/ostore.hh>** and **<ostore/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 or when either member is indexable. Calls to this macro should appear at 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**. For indexable members, use a call to the macro **os_index()**, indicating the name of the defining class's **os_backptr** member.

inv_index specifies whether the inverse member is indexable. For nonindexable members, use **os_no_index**. For indexable members, use a call to the macro **os_index()**, indicating the name of the defining class's **os_backptr** member.

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.

Required include files

To use this macro, you must include the file **<ostore/relat.hh>** after including **<ostore/ostore.hh>** and **<ostore/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 or when either member is indexable. Calls to this macro should appear at 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**. For indexable members, use a call to the macro **os_index()**, indicating the name of the defining class's **os_backptr** member, or use **os_auto_index**.

inv_index specifies whether the inverse member is indexable. For nonindexable members, use **os_no_index**. For indexable members, use a call to the macro **os_index()**, indicating the name of the defining class's **os_backptr** member, or use **os_auto_index**.

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.

Required include files

To use this macro, you must include the file **<ostore/relat.hh>** after including **<ostore/ostore.hh>**. If you also include **<ostore/coll.hh>**, include **<ostore/relat.hh>** after both **<ostore/ostore.hh>** and **<ostore/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.

Required include files

To use this macro, you must include the file **<ostore/relat.hh>** after including **<ostore/ostore.hh>** and **<ostore/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 schema 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_MACRO __declspec(dllexport)
#include <ostore/relat.hh>
```

If 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.

Required include files

To use this macro, you must include the file **<ostore/relat.hh>** after including **<ostore/ostore.hh>** and **<ostore/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.

Required include files

To use this macro, you must include the file **<ostore/relat.hh>** after including **<ostore/ostore.hh>** and **<ostore/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.

Chapter 5
C Library Interface

ObjectStore provides C functions and macros analogous to many of the functions in the ObjectStore C++ class and function libraries. This chapter presents the C library interface for ObjectStore, which allows C programs to access basic ObjectStore functionality.

Topics

Overview

ObjectStore includes a C library interface that allows access to many of ObjectStore’s features directly from C programs.

This chapter presents the ObjectStore C library interface for collections and queries. For information on the interface for other features, see the *ObjectStore C++ API Reference*.

To access the C library interface, include the following directive in your C programs:

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

Note that this header file provides access to ObjectStore’s exception facility, which provides a stock of predefined errors that can be signaled at run time. For more information, see Appendix, Predefined TIX Exceptions, on page 335.

To use ObjectStore collections, also include

```
#include <ostore/coll.h>
```

Calling the C interface from a C++ main program requires the following directives in the following order:

```
#define _PROTOTYPES

#include <ostore/ostore.hh>

extern "C" {
#include <ostore/ostore.h>
}
```

To use collections, follow this with

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

extern "C" {
#include <ostore/coll.h>
}
```

Getting Started

The building blocks of the C library interface are

- Type specifiers that you declare and allocate.
- The macro **OS_MARK_SCHEMA_TYPE**, which informs the schema generator of the structs your application uses in a persistent context.
- The macros **OS_BEGIN_TXN** and **OS_END_TXN**, which start and end a transaction, and correspond to ObjectStore's transaction statements. (All access to persistent data must take place within a transaction.)
- The function **objectstore_initialize()**, which must be executed in a process before any use of ObjectStore functionality is made.
- The allocation functions, including **os_database_alloc()** and **os_segment_alloc()**, which allocate persistent objects.
- The function **objectstore_delete(),** which corresponds to the C++ operator **delete**. You can reclaim both persistent and transient storage with the **objectstore_delete** function.
- C functions that correspond to ObjectStore's member functions and static data members.

See Building Blocks in *ObjectStore C++ API Reference* for more information.

os_backptr Functions

The C library interface contains macros for index maintenance analogous to members of the class **os_backptr** in the ObjectStore class library: **os_indexable_setvalue()** and **os_indexable_body_with_copy()**. These functions are used for index maintenance in conjunction with the macros **os_indexable_member()**, **os_indexable_body()**, **os_index()**, and **os_index_key()**. (See Chapter 4, System-Supplied Macros and User-defined Functions, on page 283 for further information.)

os_bound_query Functions

The C library interface contains functions analogous to those of the class **os_bound_query** in the ObjectStore class library.

os_bound_query_create

```
extern os_bound_query* os_bound_query_create(
    os_coll_query*,            /* the query to bind */
    os_keyword_arg_list*/* the arg list with binding for free vars */
);
```

Creates a bound query. See **os_bound_query::os_bound_query()** on page 57.

os_bound_query_delete

```
extern void os_bound_query_delete(
    os_bound_query*
);
```

Deletes the specified bound query.

os_collection Functions and Enumerators

The C library interface contains functions and enumerators analogous to those of the class **os_collection** in the ObjectStore Class Library. Programs using these functions must first call **os_collection_initialize()**, and must include **ostore/coll.h** after including **ostore/ostore.h**.

os_collection_add_index

```
extern void os_collection_add_index(
    os_collection*,/* the collection to be indexed */
    os_index_path*,/* the index path */
    unsigned int/* index options */
);
```

See **os_collection::add_index()** on page 92.

os_collection_add_index_in_seg

```
extern void os_collection_add_index_in_seg(
    os_collection*,/* the collection to be indexed */
    os_index_path*,/* the index path */
    unsigned int,/* index options */
    os_segment*/* segment of the index */
);
```

See **os_collection::add_index()** on page 92.

os_collection_bound_query

```
extern os_collection* os_collection_bound_query(
    os_collection*, /* the collection to query */
    os_bound_query*/* the query to apply */
);
```

See **os_collection::query()** on page 118.

os_collection_bound_query_exists

```
extern int os_collection_bound_query_exists(
    os_collection*,/* the collection to query */
    os_bound_query*/* the existential query to apply */
);
```

See **os_collection::exists()** on page 102.

os_collection_bound_query_pick

```
extern void* os_collection_bound_query_pick(
```

```
    os_collection*,/* the collection to query */
    os_bound_query*/* the pick query to apply */
);
```

See **os_collection::query_pick()** on page 121.

os_collection_cardinality

```
extern unsigned int os_collection_cardinality(
    os_collection*/* the collection */
);
```

See **os_collection::cardinality()** on page 97.

os_collection_change_behavior

```
extern void os_collection_change_behavior(
    os_collection*,
    unsigned int,/* new behavior flags */
    int     /* true means verify that coll meets behavior */
);
```

See **os_collection::change_behavior()** on page 97.

os_collection_change_rep

```
extern void os_collection_change_rep(
    os_collection*, /* the collection to be changed */
    unsigned int, /* the new expected size */
    os_coll_rep_descriptor*,
            /* the rep policy descriptor to change to (or 0) */
    int     /* true means retain rep policy descriptor */
);
```

See **os_collection::change_rep()** on page 98.

os_collection_clear

```
extern void os_collection_clear(
    os_collection*/* the collection to clear */
);
```

See **os_collection::clear()** on page 99.

os_collection_contains

```
extern int os_collection_contains(
    os_collection*,/* the collection */
    void*         /* the element to search for */
);
```

See **os_collection::contains()** on page 99.

os_collection_copy

```
extern void os_collection_copy(
    /* copy source elements to destination */
    os_collection*, /* destination */
    os_collection*/* source */
);
```

See **os_collection::operator =()** on page 115.

os_collection_count

```
extern unsigned int os_collection_count(
    os_collection*, /* the collection */
    void*          /* the element to count */
);
```

See **os_collection::count()** on page 99.

os_collection_create

```
extern os_collection* os_collection_create(
    os_database*,/* where to create */
    unsigned int,/* flags denoting desired behavior (or 0) */
    int,        /* expected size (or 0) */
    os_coll_rep_descriptor *,/* representation policy (or 0) */
    int         /* true means retain policy descriptor */
);
```

See **os_collection::create()** on page 99.

os_collection_create_in_cluster

```
extern os_collection* os_collection_create_in_cluster(
    os_object_cluster*,/* where to create */
    unsigned int,/* flags denoting desired behavior (or 0) */
    int,            /* expected size (or 0) */
    os_coll_rep_descriptor *,/* representation policy (or 0) */
    int             /* true means retain policy descriptor */
);
```

See **os_collection::create()** on page 99.

os_collection_create_in_seg

```
extern os_collection* os_collection_create_in_seg(
    os_segment*, /* where to create */
    unsigned int,/* flags denoting desired behavior (or 0) */
    int,        /* expected size (or 0) */
    os_coll_rep_descriptor *,/* representation policy (or 0) */
    int         /* true means retain policy descriptor */
);
```

See **os_collection::create()** on page 99.

os_collection_create_near

```
extern os_collection* os_collection_create_near(
    void*,          /* where to create */
    unsigned int,/* flags denoting desired behavior (or 0) */
    int,            /* expected size (or 0) */
    os_coll_rep_descriptor *,/* representation policy (or 0) */
    int             /* true means retain policy descriptor */
);
```

See **os_collection::create()** on page 99.

os_collection_delete

```
extern void os_collection_delete(
    os_collection*/* the collection to delete */
);
```

Deletes the specified collection.

os_collection_difference

```
extern void os_collection_difference(
    /* subtract source elements from destination */
    os_collection*,/* destination */
    os_collection*/* source */
);
```

See **os_collection::operator -()** on page 117.

os_collection_drop_index

```
extern void os_collection_drop_index(
    os_collection*,/* the collection with the index */
    os_index_path*/* the index to drop */
);
```

See **os_collection::drop_index()** on page 102.

os_collection_empty

```
extern int os_collection_empty(
    os_collection*/* check if the collection is empty */
);
```

See **os_collection::empty()** on page 102.

os_collection_equal

```
extern int os_collection_equal(
    os_collection*,
```

```
    os_collection*
);
```

See **os_collection::operator ==()** on page 113.

os_collection_get_behavior

```
extern unsigned int os_collection_get_behavior(
    /* return flags denoting behavior */
    os_collection*
);
```

See **os_collection::get_behavior()** on page 105.

os_collection_get_rep

```
extern os_coll_rep_descriptor* os_collection_get_rep(
    os_collection*
);
```

See **os_collection::get_rep()** on page 105.

os_collection_greater_than

```
extern int os_collection_greater_than(
    os_collection*,
    os_collection*
);
```

See **os_collection::operator >()** on page 114.

os_collection_greater_than_or_equal

```
extern int os_collection_greater_than_or_equal(
    os_collection*,
    os_collection*
);
```

See **os_collection::operator >=()** on page 114.

os_collection_has_index

```
extern int os_collection_has_index(
    os_collection*,/* the collection to look for an index on */
    os_index_path*,/* the index to look for */
    int            /* true if looking for an ordered index */
);
```

See **os_collection::has_index()** on page 106.

os_collection_initialize

```
extern void os_collection_initialize();
```

See **os_collection::initialize()** on page 107.

os_collection_insert

```
extern void os_collection_insert(
    os_collection*,/* the collection */
    void*          /* the element to insert */
);
```

See **os_collection::insert()** on page 107.

os_collection_insert_after_cursor

```
extern void os_collection_insert_after_cursor(
    os_collection*,
    void*,
    os_cursor*
);
```

See **os_collection::insert_after()** on page 107.

os_collection_insert_after_position

```
extern void os_collection_insert_after_position(
    os_collection*,
    void*,
    unsigned int
);
```

See **os_collection::insert_after()** on page 107.

os_collection_insert_before_cursor

```
extern void os_collection_insert_before_cursor(
    os_collection*,
    void*,
    os_cursor*
);
```

See **os_collection::insert_before()** on page 108.

os_collection_insert_before_position

```
extern void os_collection_insert_before_position(
    os_collection*,
    void*,
    unsigned int
);
```

See **os_collection::insert_before()** on page 108.

os_collection_insert_first

```
extern void os_collection_insert_first(
    os_collection*,/* the collection */
    void*          /* the element to insert */
);
```

See **os_collection::insert_first()** on page 108.

os_collection_insert_last

```
extern void os_collection_insert_last(
    os_collection*,/* the collection */
    void*          /* the element to insert */
);
```

See **os_collection::insert_last()** on page 109.

os_collection_intersect

```
extern void os_collection_intersect(
    os_collection*,/* destination */
    os_collection*/* source */
);
```

See **os_collection::operator &()** on page 116.

os_collection_less_than

```
extern int os_collection_less_than(
    os_collection*,
    os_collection*
);
```

See **os_collection::operator <()** on page 114.

os_collection_less_than_or_equal

```
extern int os_collection_less_than_or_equal(
    os_collection*,
    os_collection*
);
```

See **os_collection::operator <=()** on page 114.

os_collection_not_equal

```
extern int os_collection_not_equal(
    os_collection*,
    os_collection*
);
```

See **os_collection::operator !=()** on page 113.

os_collection_only

```
extern void* os_collection_only(
    os_collection*
);
```

See **os_collection::only()** on page 111.

os_collection_ordered_equal

```
extern int os_collection_ordered_equal(
    os_collection*,
    os_collection*
);
```

See **os_collection::operator ==()** on page 113.

os_collection_pick

```
extern void* os_collection_pick(
    os_collection*
);
```

See **os_collection::pick()** on page 117.

os_collection_query

```
extern os_collection* os_collection_query(
    os_collection*,/* the collection to query */
    char*,    /* the string denoting the element type */
    char*,    /* the string denoting the query expression */
    os_database*,/* the database from which to get the schema */
    char*,    /* name of file (for error printing) or 0 */
    unsigned int/* line number in file (for error printing) or 0 */
);
```

See **os_collection::query()** on page 118.

os_collection_query_exists

```
extern int os_collection_query_exists(
    os_collection*,/* the collection to query */
    char*,    /* the string denoting the element type */
    char*,    /* the string denoting the query expression */
    os_database*,/* the database from which to get the schema */
    char*,    /* name of file (for error printing) or 0 */
    unsigned int /* line number in file (for error printing) or 0 */
);
```

See **os_collection::exists()** on page 102.

os_collection_query_pick

```
extern void* os_collection_query_pick(
    os_collection*,/* the collection to query */
    char*,     /* the string denoting the element type */
    char*,     /* the string denoting the query expression */
    os_database*,/* the database from which to get the schema */
    char*,     /* name of file (for error printing) or 0 */
    unsigned int /* line number in file (for error printing) or 0 */
);
```

See **os_collection::query_pick()** on page 121.

os_collection_remove

```
extern int os_collection_remove(
    os_collection*,/* the collection */
    void*      /* the element to remove */
);
```

See **os_collection::remove()** on page 124.

os_collection_remove_at_cursor

```
extern void os_collection_remove_at_cursor(
    os_collection*,
    os_cursor*
);
```

See **os_collection::remove_at()** on page 124.

os_collection_remove_at_position

```
extern void os_collection_remove_at_position(
    os_collection*,
    unsigned int
);
```

See **os_collection::remove_at()** on page 124.

os_collection_remove_first

```
extern void* os_collection_remove_first(
    os_collection*
);
```

See **os_collection::remove_first()** on page 124.

os_collection_remove_last

```
extern void* os_collection_remove_last(
    os_collection*
);
```

See **os_collection::remove_last()** on page 125.

os_collection_replace_at_cursor

```
extern void* os_collection_replace_at_cursor(
    os_collection*,
    void*,
    os_cursor*
);
```

See **os_collection::replace_at()** on page 125.

os_collection_replace_at_position

```
extern void* os_collection_replace_at_position(
    os_collection*,
    void*,
    unsigned int
);
```

See **os_collection::replace_at()** on page 125.

os_collection_retrieve_at_cursor

```
extern void* os_collection_retrieve_at_cursor(
    os_collection*,
    os_cursor*
);
```

See **os_collection::retrieve()** on page 126.

os_collection_retrieve_at_position

```
extern void* os_collection_retrieve_at_position(
    os_collection*,
    unsigned int
);
```

See **os_collection::retrieve()** on page 126.

os_collection_retrieve_first

```
extern void* os_collection_retrieve_first(
    os_collection*
);
```

See **os_collection::retrieve_first()** on page 126.

os_collection_retrieve_last

```
extern void* os_collection_retrieve_last(
    os_collection*
);
```

See **os_collection::retrieve_last()** on page 126.

os_collection_union

```
extern void os_collection_union(
    /* union source elements into destination */
    os_collection*,/* destination */
    os_collection*/* source */
);
```

See **os_collection::operator |()** on page 115.

os_coll_query Functions

The C library interface contains functions analogous to those of the class **os_coll_query** in the ObjectStore Class Library.

os_coll_query_create

```
extern os_coll_query *os_coll_query_create(
    char*,    /* string denoting the element type */
    char*,    /* string denoting the query expression */
    os_database*,/* schema for query interpretation */
    os_int32,/* true means cache the query in db */
    char*,    /* file name (for error messages) or 0 */
    unsigned/* line number in file or 0 */
);
```

Creates a query. See **os_coll_query::create()** on page 130.

os_coll_query_create_exists

```
extern os_coll_query *os_coll_query_create_exists(
    char*,        /* string denoting the element type */
    char*,        /* string denoting the query expression */
    os_database*,/* schema for query interpretation */
    os_int32,/* true means cache the query persistently in db */
    char*,        /* file name (for error messages) or 0 */
    unsigned/* line number in file or 0 */
);
```

Creates an existential query. See **os_coll_query::create_exists()** on page 134.

os_coll_query_create_exists_in_seg

```
extern os_coll_query *os_coll_query_create_exists_in_seg(
    char*,    /* string denoting the element type */
    char*,    /* string denoting the query expression */
    os_segment*,/* schema for query interpretation */
    os_int32,/* true means cache the query in db */
    char*,    /* file name (for error messages) or 0 */
    unsigned/* line number in file or 0 */
);
```

Creates an existential query in the specified segment. See **os_coll_query::create_exists()** on page 134.

os_coll_query_create_in_seg

```
extern os_coll_query *os_coll_query_create_in_seg(
    char*,    /* string denoting the element type */
```

```
    char*,    /* string denoting the query expression */
    os_segment*,/* schema for query interpretation */
    os_int32,/* true means cache the query persistently in seg */
    char*,    /* file name (for error messages) or 0 */
    unsigned/* line number in file or 0 */
);
```

Creates a query in the specified segment. See **os_coll_query::create()** on page 130.

os_coll_query_create_pick

```
extern os_coll_query *os_coll_query_create_pick(
    char*,    /* string denoting the element type */
    char*,    /* string denoting the query expression */
    os_database*,/* schema for query interpretation */
    os_int32,/* true means cache the query persistently in db */
    char*,    /* file name (for error messages) or 0 */
    unsigned/* line number in file or 0 */
);
```

Creates a single-element query. See **os_coll_query::create_pick()** on page 135.

os_coll_query_create_pick_in_seg

```
extern os_coll_query *os_coll_query_create_pick_in_seg(
    char*,    /* string denoting the element type */
    char*,    /* string denoting the query expression */
    os_segment*,/* schema for query interpretation */
    os_int32,/* true means cache the query in seg */
    char*,    /* file name (for error messages) or 0 */
    unsigned/* line number in file or 0 */
);
```

Creates a single-element query in the specified segment. See **os_coll_query::create_pick()** on page 135.

os_coll_rep_descriptor Functions

The C library interface contains functions analogous to those of the class **os_coll_rep_descriptor** in the ObjectStore Class Library.

os_coll_rep_descriptor

extern os_coll_rep_descriptor* os_coll_get_packed_list_rep_ descriptor();

Returns an **os_packed_list** rep descriptor.

extern os_coll_rep_descriptor* os_coll_get_ptr_bag_list_rep_ descriptor();

Returns an **os_ptr_bag** rep descriptor.

extern os_coll_rep_descriptor* os_coll_get_ptr_hash_rep_ descriptor();

Returns an **os_ptr_hash** rep descriptor.

extern os_coll_rep_descriptor* os_coll_get_tinyarray_rep_ descriptor();

Returns an **os_tinyarray** rep descriptor.

os_coll_rep_descriptor_allowed_behavior

```
extern unsigned os_coll_rep_descriptor_allowed_behavior(
    os_coll_rep_descriptor*
    /* return the behavior that this rep supports */
);
```

Returns a bit pattern indicating the behavior supported by the specified rep descriptor.

os_coll_rep_descriptor_copy

```
extern os_coll_rep_descriptor* os_coll_rep_descriptor_copy(
    os_coll_rep_descriptor,
    /* make a copy of this rep descriptor */
    os_segment * /* in this segment */
);
```

Copies the specified descriptor. See **os_coll_rep_descriptor::copy()** on page 143.

os_coll_rep_descriptor_get_grow

```
extern os_coll_rep_descriptor* os_coll_rep_descriptor_get_grow(
    os_coll_rep_descriptor*
```

```
    /* return this descriptor's grow-into descriptor */
);
```

Returns the rep descriptor that becomes active when the specified rep descriptor's maximum cardinality is exceeded.

os_coll_rep_descriptor_get_max_size

```
extern unsigned os_coll_rep_descriptor_get_max_size(
    os_coll_rep_descriptor*
    /* return this descriptor's max size */
);
```

Returns the upper bound of the specified rep descriptor's associated cardinality range.

os_coll_rep_descriptor_get_min_size

```
extern unsigned os_coll_rep_descriptor_get_min_size(
    os_coll_rep_descriptor*
    /* return this descriptor's min size */
);
```

Returns the lower bound of the specified rep descriptor's associated cardinality range.

os_coll_rep_descriptor_get_shrink

```
extern os_coll_rep_descriptor* os_coll_rep_descriptor_get_shrink(
    os_coll_rep_descriptor*
    /* return this descriptor's shrink-into descriptor */
);
```

Returns the rep descriptor that becomes active when the specified rep descriptor's minimum cardinality threshold is passed.

os_coll_rep_descriptor_required_behavior

```
extern unsigned os_coll_rep_descriptor_required_behavior(
    os_coll_rep_descriptor*
    /* return the behavior that this rep requires */
);
```

Returns a bit pattern indicating the behavior required of collections with the specified representation.

os_cursor Functions

The C library interface contains functions analogous to those of the class **os_cursor** in the ObjectStore Class Library.

os_cursor_copy

```
extern void os_cursor_copy(
    os_cursor*, /* destination */
    os_cursor* /* source */
);
```

Copies **source** to **destination**.

os_cursor_create

```
extern os_cursor* os_cursor_create(
    os_collection*, /* create a cursor over this collection */
    int /* true means allow for updates during iteration */
);
```

Creates a cursor for the specified collection. See **os_cursor::os_cursor()** on page 156.

os_cursor_create_in_cluster(

```
extern os_cursor* os_cursor_create_in_cluster(
    os_object_cluster*, /* create in this cluster */
    os_collection*,    /* create a cursor over this collection */
    os_int32   /* bitmask option: forward/reverse, order_by_address */
        /* safe/unsafe etc, enums */
        /* safe allow for updates during iteration */
);
```

See **os_cursor::os_cursor()** on page 156.

os_cursor_create_in_db

```
extern os_cursor* os_cursor_create_in_db(
    os_database*,  /* create in this database */
    os_collection*, /* create a cursor over this collection */
    os_int32     /* bitmask: forward/reverse, order_by_address */
        /* safe/unsafe enums */
        /* safe allow for updates during iteration */
);
```

See **os_cursor::os_cursor()** on page 156.

os_cursor_create_in_seg

```
extern os_cursor* os_cursor_create_in_seg(
```

```
    os_segment*, /* create in this segment */
    os_collection*, /* create a cursor over this collection */
    os_int32  /* bitmask option: forward/reverse, order_by_address */
        /* safe/unsafe etc, enums */
        /* safe allow for updates during iteration */
);
```

See **os_cursor::os_cursor()** on page 156.

os_cursor_create_near

```
extern os_cursor* os_cursor_create_near(
    void*,       /* where to create this */
    os_collection*, /* create a cursor over this collection */
    os_int32   /* bitmask: forward/reverse, order_by_address */
    /* safe/unsafe, etc enums */
    /* safe allow for updates during iteration */
);
```

See **os_cursor::os_cursor()** on page 156.

os_cursor_create_options

```
extern os_cursor* os_cursor_create_options(
    os_collection*, /* create a cursor over this collection */
    os_int32   /* bitmask: forward/reverse, order_by_address */
        /* safe/unsafe etc, enums */
        /* safe allow for updates during iteration */
);
```

See **os_cursor::os_cursor()** on page 156.

os_cursor_delete

```
extern void os_cursor_delete(
    os_cursor*
);
```

Destroys the specified cursor and frees its associated memory.

os_cursor_first

```
extern void* os_cursor_first(
    os_cursor* /* put the cursor on the first element and return it */
);
```

See **os_cursor::first()** on page 155.

os_cursor_insert_after

```
extern void os_cursor_insert_after(
    os_cursor*,
     /* insert after this position, in the cursor's collection */
```

```
    void*     /* element to insert */
);
```

See **os_cursor::insert_after()** on page 155.

os_cursor_insert_before

```
extern void os_cursor_insert_before(
    os_cursor*,
    *insert before this position, in the cursor's collection */
    void*          /* element to insert */
);
```

See **os_cursor::insert_before()** on page 155.

os_cursor_last

```
extern void* os_cursor_last(
    os_cursor* /* put the cursor on the last element and return it */
);
```

See **os_cursor::last()** on page 155.

os_cursor_more

```
extern int os_cursor_more(
    os_cursor* /* return true if this cursor is not null */
);
```

See **os_cursor::more()** on page 156.

os_cursor_next

```
extern void* os_cursor_next(
    os_cursor* /* put the cursor on the next element and return it */
);
```

See **os_cursor::next()** on page 156.

os_cursor_null

```
extern int os_cursor_null(
    os_cursor* /* return true if this cursor is null */
);
```

See **os_cursor::null()** on page 156.

os_cursor_ordered_create

```
extern os_cursor* os_cursor_ordered_create(
    os_collection*,  /* create a cursor over this collection */
    os_index_path*, /* path to codify order of an ordered iteration */
    os_int32     /* true means allow for updates during iteration */
```

```
);
```

See **os_cursor::os_cursor()** on page 156.

os_cursor_ordered_create_in_cluster

```
extern os_cursor* os_cursor_ordered_create_in_cluster(
    os_object_cluster*, /* create cursor in this cluster */
    os_collection*,   /* create a cursor over this collection */
    os_index_path*,    /* path to codify order of an ordered iteration */
    os_int32 /* bitmask option: forward/reverse, order_by_address */
        /* safe/unsafe etc, enums */
        /* safe allow for updates during iteration */
);
```

See **os_cursor::os_cursor()** on page 156.

os_cursor_ordered_create_in_db

```
extern os_cursor* os_cursor_ordered_create_in_db(
    os_database*,    /* create in this database */
    os_collection*,   /* create a cursor over this collection */
    os_index_path*, /* path to codify order of an ordered iteration */
    os_int32 /* bitmask option: forward/reverse, order_by_address */
        /* safe/unsafe etc, enums */
        /* safe allow for updates during iteration */
);
```

See **os_cursor::os_cursor()** on page 156.

os_cursor_ordered_create_in_seg

```
extern os_cursor* os_cursor_ordered_create_in_seg(
    os_segment*, /* create in this segment */
    os_collection*, /* create a cursor over this collection */
    os_index_path*, /* path to codify order of an ordered iteration */
    os_int32 /* bitmask option: forward/reverse, order_by_address */
        /* safe/unsafe etc, enums */
        /* safe allow for updates during iteration */
);
```

See **os_cursor::os_cursor()** on page 156.

os_cursor_ordered_create_near

```
extern os_cursor* os_cursor_ordered_create_near(
    void *,       /* create cursor in this */
    os_collection*,   /* create a cursor over this collection */
    os_index_path*, /* path to codify order of an ordered iteration */
    os_int32 /* bitmask: forward/reverse, order_by_address */
        /* safe/unsafe, etc enums */
        /* safe allow for updates during iteration */
```

```
);
```

See **os_cursor::os_cursor()** on page 156.

os_cursor_ordered_create_options

```
extern os_cursor* os_cursor_ordered_create_options(
    os_collection*,    /* create a cursor over this collection */
    os_index_path*,  /* path to codify order of an ordered iteration */
    os_int32 /* bitmask option: forward/reverse, order_by_address */
        /* safe/unsafe etc, enums */
        /* safe allow for updates during iteration */
);
```

See **os_cursor::os_cursor()** on page 156.

os_cursor_previous

```
extern void* os_cursor_previous(
    os_cursor*
     /* put the cursor on the previous element and return it */
);
```

See **os_cursor::previous()** on page 159.

os_cursor_remove_at

```
extern void os_cursor_remove_at(
    os_cursor*
     /* remove the element in the collection at this position */
);
```

See **os_cursor::remove_at()** on page 159.

os_cursor_retrieve

```
extern void* os_cursor_retrieve(
    os_cursor* /* return the element at the current cursor position */
);
```

See **os_cursor::retrieve()** on page 159.

os_cursor_valid

```
extern int os_cursor_valid(
    os_cursor* /* return true if this cursor is at an element */
);
```

See **os_cursor::valid()** on page 160.

os_index_path Functions

The C library interface contains functions analogous to those of the class **os_index_path** in the ObjectStore Class Library.

os_index_path_create

```
extern os_index_path* os_index_path_create(
    char*,     /* string denoting element type (start of path) */
    char*,     /* string denoting the path */
    os_database*/* database in which to create path */
);
```

Creates an index path. See **os_index_path::create()** on page 179.

os_index_path_create_in_seg

```
extern os_index_path* os_index_path_create_in_seg(
    char*,     /* string denoting element type (start of path) */
    char*,     /* string denoting the path */
    os_segment*/* segment in which to create path */
);
```

Creates an index path. See **os_index_path::create()** on page 179.

os_index_path_delete

```
extern void os_index_path_delete(
    os_index_path*
);
```

Deletes an index path. See **os_index_path::destroy()** on page 181.

Appendix
Predefined TIX Exceptions

This section contains information on significant predefined exceptions. These exceptions are defined in **client.hh** and **ostore.h**, so they are automatically available to your programs.

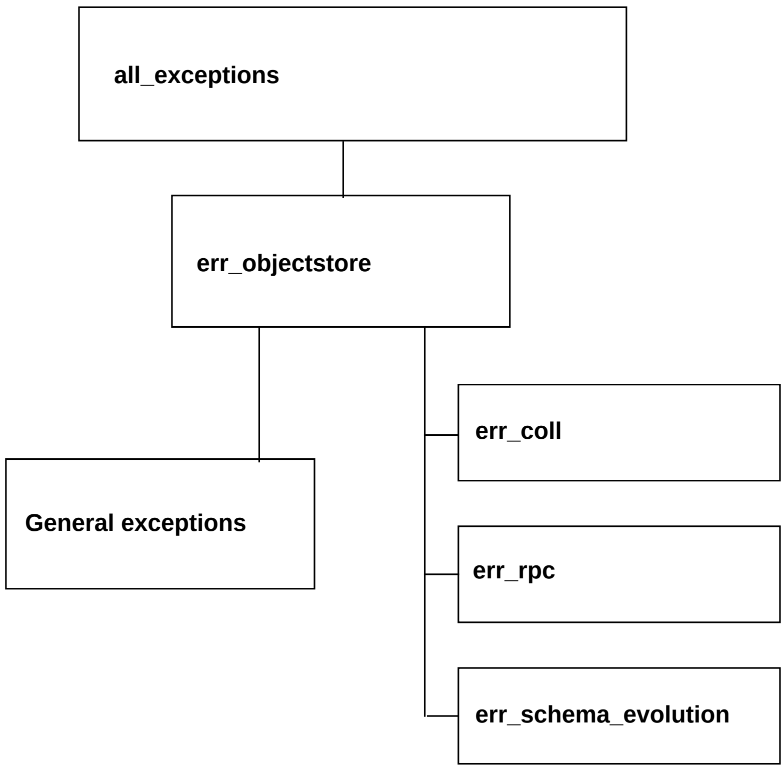
Topics

Parent Exceptions

The following are *parents* in the exceptions object tree hierarchy. They are never signaled directly, but it can be useful to set up handlers for them in order to catch an entire set of errors.

ObjectStore exception inheritance hierarchy

The hierarchy is arranged as follows:



- Every TIX exception is a descendant of **all_exceptions**.
- Every TIX exception that is signaled by ObjectStore itself is a child of **err_objectstore**, which is a child of **all_exceptions**.
- Every TIX exception signaled from the remote procedure call (RPC) mechanism (which ObjectStore uses for all its network communications) is a child of **err_rpc**, which is a child of **err_objectstore**.

The ObjectStore exception facility itself is presented in Appendix A, Exception Facility, in *ObjectStore C++ API Reference*.

Predefined Exceptions

Collection Exceptions

The following exceptions descend from **err_coll**, which is a descendent of **err_objectstore.**

err_am. Error using indexes, for example, an attempt to add an index where a class mentioned in the path serving as index key cannot be found in the schema of the database containing the index (or the application schema, if the index is transient).

Can be signaled by:

- os_collection::add_index()
- os_coll_range::os_coll_range()

err_coll. The parent of all collection exceptions.

err_coll_ambiguous.

not found in coll_class

err_coll_behavior_inconsistency. The representation policy was semantically inconsistent with regard to the collection object.

not found in coll_class.

err_coll_cannot_grow_collection. An attempt was made to grow a collection that could not be grown, usually because the **grow_by** or the **grow_at** parameter to collection creation specified no growth.

err_coll_cannot_mutate_collection. A collection could not be mutated into an alternate representation.

err_coll_dangling_pointer. A dangling pointer from a collection to a deleted object was detected, due to the presence of **os_backptr** during deletion of the containing object.

err_coll_duplicates. An attempt was made to duplicate an element in a collection.

os_collection::allow_duplicates

os_collection::change_behavior

os_collection::insert

os_collection::insert_after

os_Collection::insert

os_Collection::insert_after

os_Collection::insert_before

os_Collection::insert_first

err_coll_empty. The protocol expected a nonempty set, but was used on an empty set instead.

os_Array::create (pick)

os_array::create

err_coll_evolve. The root exception for collection evolution.

err_coll_evolve_not_implemented_yet. The unimplemented part of collection evolution.

err_coll_illegal_arg. An actual argument used in the collection protocol failed validation. The text of the report contains details regarding the specific argument.

err_coll_illegal_cast. An illegal cast operation was attempted.

err_coll_illegal_cursor.

err_coll_illegal_query_expression. Syntax / semantic analysis of the query text resulted in an error.

err_coll_illegal_update. An attempt was made to update a **const** collection through a cursor.

err_coll_internal. This exception is used to signal internal collection errors.

err_coll_internal_list. An error occurred in an internal list.

err_coll_none_qualifying. An error occurred in index lookup during scan.

err_coll_not_implemented_yet. For as-yet-unimplemented collection features.

err_coll_not_singleton. **os_collection::only** expected a singleton set, but the **cardinality()** != 1.

err_coll_not_ordered. The operation required that the collection be ordered, but it was not.

err_coll_not_supported. An attempt was made to use a collection subtype-specific protocol that was not supported by this particular subtype.

err_coll_null_cursor. The protocol expected a nonnull cursor for the particular operation.

err_coll_nulls. An attempt was made to insert a null element into a collection.

os_collection::allow_nulls

err_coll_out_of_range. A collection was accessed using an out-of-bounds array subscript.

err_coll_path_interp. An error in path interpretation occurred.

err_coll_query_bind. The query had free references, but these references were not bound at the * of the query. The report identifies the unbound variables.

err_coll_query_evaluate. An error occurred during evaluation of a query.

err_coll_scan. An error in scan occurred.

err_cursor. An error was made in cursor maintenance.

err_cursor_ambiguous. An error was made in cursor order specification.

err_cursor_not_implemented_yet. Unimplemented feature.

err_cursor_internal. An error in ordered iteration occurred.

err_illegal_index_path. An error occurred during translation of an index path expression.

err_index. An error occurred in an index.

err_index_duplicate_key. The uniqueness constraint on an index was violated.

err_index_evolve. An error occurred during the evolution of an index.

err_index_not_implemented_yet. For as-yet-unimplemented index evolution features.

err_index_invalid_ordering. An index was ordered in an invalid way.

err_index_wrong_kind. An unordered index was used for ordered iteration.

err_null_cursor. An attempt was made to operate on a null cursor.

err_object_init. Derived from **err_objectstore**, this exception can be caught by the application (as **err_objectstore**). This is the exception generated by all the error conditions that

- Are not in a transaction
- Have no **type_name** provided with a transient instance
- Have a **type_name** mismatch with a persistent instance
- Could not find schema information for type
- Are called with an embedded object (persistent only)

err_open_iteration. An iteration open on mapping being deleted.

err_pset_no_cursor. Error in **_Pset** iteration.

Index

D

F

G

N

O

P

Q

R

S

U

V