OBJECTSTORE

C++ API REFERENCE

RELEASE 5.1

March 1998

ObjectStore C++ API Reference

ObjectStore Release 5.1, March 1998

ObjectStore, Object Design, the Object Design logo, LEADERSHIP BY DESIGN, and Object Exchange are registered trademarks of Object Design, Inc. ObjectForms and Object Manager are trademarks of Object Design, Inc.

All other trademarks are the property of their respective owners.

Copyright © 1989 to 1998 Object Design, Inc. All rights reserved. Printed in the United States of America. Except as permitted under the Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of the publisher.

COMMERCIAL ITEM — The Programs are Commercial Computer Software, as defined in the Federal Acquisition Regulations and Department of Defense FAR Supplement, and are delivered to the United States Government with only those rights set forth in Object Design's software license agreement.

Data contained herein are proprietary to Object Design, Inc., or its licensors, and may not be used, disclosed, reproduced, modified, performed or displayed without the prior written approval of Object Design, Inc.

This document contains proprietary Object Design information and is licensed for use pursuant to a Software License Services Agreement between Object Design, Inc., and Customer.

The information in this document is subject to change without notice. Object Design, Inc., assumes no responsibility for any errors that may appear in this document.

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

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

Contents

Preface ix

Chapter 1 Introduction 1

ObjectStore Database Services 2

Chapter 2 Class Library 7

objectstore 11
os_access_modifier 44
os_address_space_marker 45
os_anonymous_indirect_type 49
os_app_schema 50
os_application_schema_info 51
os_array_type 52
os_base_class 54
os_class_type 57
os_comp_schema 70
os_database 71
os_database_reference 101
os_database_root 103
os_database_schema 105
os_dbutil 106
os_DLL_finder 128
os_DLL_schema_info 130
os_dynamic_extent 132
os_enum_type 135
os_enumerator_literal 137

os_evolve_subtype_fun_binding 138
os_failover_server 139
os_field_member_variable 141
os_function_type 142
os_indirect_type 144
os_instantiated_class_type 145
os_integral_type 146
os_literal 148
os_literal_template_actual_arg 151
os_lock_timeout_exception 152
os_member 154
os_member_function 159
os_member_namespace 164
os_member_type 165
os_member_variable 166
os_mop 170
os_named_indirect_type 174
os_namespace 175
os_notification 176
os_object_cluster 186
os_object_cursor 189
os_pathname_and_segment_number 191
os_pointer_literal 192
os_pointer_to_member_type 193
os_pointer_type 194
os_pragma 195
os_pvar 196
os_rawfs_entry 199
os_real_type 202
os_Reference 203
os_reference 209
os_Reference_local 215
os_reference_local 220
os_Reference_protected 224
os_reference_protected 231
os_Reference_protected_local 237

os_reference_protected_local 242
os_Reference_this_DB 247
os_reference_this_DB 252
os_Reference_transient 257
os_reference_transient 262
os_reference_type 266
os_relationship_member_variable 267
os_retain_address 268
os_schema 270
os_schema_evolution 272
os_schema_handle 289
os_schema_info 293
os_schema_install_options 294
os_segment 295
os_segment_access 307
os_server 311
os_str_conv 314
os_subscription 318
os_template 321
os_template_actual_arg 323
os_template_formal_arg 325
os_template_instantiation 327
os_template_type_formal 329
os_template_value_formal 330
os_transaction 331
os_transaction_hook 339
os_transformer_binding 341
os_type 342
os_type_template 354
os_type_template_actual_arg 355
os_type_type 356
os_typed_pointer_void 357
os_typespec 358
os_void_type 363
os_with_mapped 364

Chapter 3 System-Supplied Global Functions 365

::operator delete() 366
::operator new() 367
::os_fetch() 370
::os_fetch_address(). 373
::os_store() 374

Chapter 4 System-Supplied Macros. 377

OS_BEGIN_TXN(),
OS_BEGIN_TXN_NAMED(), and
OS_END_TXN() 378
OS_ESTABLISH_FAULT_HANDLER and
OS_END_FAULT_HANDLER. 383
OS_MARK_QUERY_FUNCTION(). 385
OS_MARK_SCHEMA_TYPE()
OS_MARK_SCHEMA_TYPESPEC() 386
Component Schema Macros. 387

Chapter 5 User-Supplied Functions. 389

Discriminant Functions. 390

Chapter 6 C Library Interface 393

Overview 395
Getting Started 397
Building Blocks 398
objectstore Functions. 402
os_database Functions 410
os_database_reference Functions 422
os_database_root Functions 423
os_dbutil Functions 424
os_keyword_arg Functions. 433
os_keyword_arg_list Functions. 436
os_lock_timeout_exception Functions 437
Metaobject Protocol Functions 439
os_notification Functions 500

os_object_cluster Functions 505
os_object_cursor Functions. 507
os_pvar Functions. 509
os_rawfs_entry Functions. 510
os_segment Functions. 513
os_segment_access Functions 520
os_server Functions 521
os_transaction Functions 523
os_typespec Functions 526
Reference Functions 529
Schema Evolution Functions 543
TIX Functions 552

Appendix A Exception Facility 555

Macros 556
Classes 559
Sample TIX Exception-Handling Routine 566

Appendix B Predefined TIX Exceptions 569

Parent Exceptions 570
General ObjectStore Exceptions 571
Schema Evolution Exceptions. 579
RPC Exceptions 581
Component Schema Exceptions. 583

Index. 585

Preface

Purpose

The *ObjectStore C++ API Reference* provides a reference on the core C++ programming interface to ObjectStore. It is supplemented by the *ObjectStore Collections C++ API Reference*, which describes the programming interface for using ObjectStore collections, queries, and indexes.

Audience

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

Scope

Information in this book assumes that ObjectStore is installed and configured. This book supports ObjectStore Release 5.1.

How This Book Is Organized

The manual has seven chapters and an appendix.

Introduction

The introduction describes the ObjectStore database services. See Chapter 1, Introduction, on page 1.

Class library

The class library chapter describes the system-supplied C++ classes, whose members and enumerators provide an interface to database features. The classes are listed alphabetically by class name. Within the entry for each class, the class's members, as well as enumerators defined within the class's scope, are listed alphabetically. See Chapter 2, Class Library, on page 7.

System-supplied global functions

Some system-supplied interface functions are not members of any class, but are global C++ functions (**operator new()** and **operator delete()**). These are listed alphabetically in Chapter 3, System-Supplied Global Functions, on page 365.

System-supplied macros

Some ObjectStore functionality is accessed through the use of macros (such as those for starting and ending transactions). These are listed alphabetically in Chapter 4, System-Supplied Macros, on page 377.

User-supplied functions

Access to database services sometimes requires support from user-defined functions. The required functions are listed alphabetically in Chapter 5, User-Supplied Functions, on page 389.

C library interface

In addition to C++ class and function libraries, ObjectStore provides C functions and macros analogous to those provided in the C++ libraries so that it is possible for C programs to access all ObjectStore functionality. These are listed in Chapter 6, C Library Interface, on page 393.

Exception facility

ObjectStore uses an exception facility added onto C++ by Object Design, Inc. An appendix is included that provides an account of signaling and handling exceptions. See Appendix A, Exception Facility, on page 555.

Predefined exceptions

This appendix is an alphabetical listing of the predefined exceptions that can be signaled by ObjectStore at run time. See Appendix B, Predefined TIX Exceptions, on page 569.

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.

Convention	*Meaning*
UNIX ~~UNIX~~	Indicates that the operating system named inside the circle supports or does not support the feature being discussed.

ObjectStore Release 5.1 Documentation

The ObjectStore Release 5.1 documentation is chiefly distributed online 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 online 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 on page 76 of reference manual

You can also fax your comments to 781.674.5440.

Chapter 1
Introduction

This document describes the application programming interface to the functionality provided by the ObjectStore object-oriented database management system. ObjectStore seamlessly integrates the C and C++ programming language with the database services required by complex, data-intensive applications. These services include support for the following:

Persistence	Provision of a central repository for information that persists beyond the lifetime of the process that recorded it
Query processing	Support for associative data retrieval, such as lookup by name or ID number
Integrity control	Services that maintain data consistency based on specified database constraints
Access control	Services that protect data against unauthorized access
Concurrency control	Services that allow shared, concurrent data access
Fault tolerance	Services that protect data consistency and prevent data corruption even in the face of system crashes or network failures
Dynamic schema access	Services that allow the programmatic manipulation of database schema information
Schema evolution	Services that support schema change and automatic modification of database objects to conform to new schemas
Data compaction	Services that support data reorganization to eliminate fragmentation

ObjectStore Database Services

This section summarizes the functionality of each ObjectStore database service.

Persistence

ObjectStore provides direct, transparent access to persistent data from within C++ programs. No explicit database reads or writes are required, and persistence is entirely orthogonal to type. This means that the same type can have both persistent and nonpersistent instances, and the same function can take both persistent and nonpersistent arguments. Moreover, the instances of any built-in C++ type can be designated as persistent.

Persistent allocation is performed with an overloading of the C++ **new** operator. This variant of **new** allows for the specification of clustering information, so that objects that should exhibit locality of reference can be clustered into the same database *segment* or *object cluster*. Segments are variable-size portions of database storage that can be used as the unit of transfer to and from the database. Object clusters are fixed-size portions of database storage that live within segments.

ObjectStore supplies a class that provides the ability to name an object to serve as a database entry point. This class also provides a function allowing the entry point object to be looked up by its name, so that it can serve as a starting point for navigational or query access. Navigational access is performed by following pointers contained in data structure fields, just as would be done in a regular, nondatabase C or C++ program. Pointer dereferencing causes transparent database retrievals when needed. Query access is described below.

A single ObjectStore application can open several databases at a time. In addition, several applications can access the same database concurrently, as described in the section on *Transactions and Concurrency Control*, below.

Query processing

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.

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.

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. By using these indexes, implemented as B-trees or hash tables, the number of objects examined in response to a query can be minimized. 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.

See *ObjectStore Collections C++ API Reference* for more information on collections, queries, and indexes.

Integrity control

Many design applications create and manipulate large amounts of complex persistent data. Frequently, this data is jointly accessed by a set of cooperative applications, each of which carries the data through some well-defined transformation. Because the data is shared, and because it is more permanent and more valuable than any particular run of an application, maintaining the data's integrity becomes a major concern and requires special database support.

In addition to the integrity control provided by compile-time type checking, ObjectStore provides facilities to help deal with some of the most common integrity maintenance problems.

One integrity control problem concerns pairs of data members that are used to model binary relationships. A binary relationship, such as the part/subpart relationship, for example, can be modeled by a pair of data members, *parent_part* and *child_part*. Modeling the relationships with both these data members has the advantage of allowing navigation both up and down the parts hierarchy. However, the data members must be kept in a consistent state with respect to one another: one object is the parent of another if and only if the other is a child of the first. This integrity constraint can be enforced by declaring the two data members as *inverses* of one another. ObjectStore automatically implements the constraint as an update dependency.

Another integrity control problem concerns *illegal pointers*. ObjectStore can dynamically detect pointers from persistent to transient memory, as well as cross-database pointers from segments specified by the user to disallow such pointers. ObjectStore's schema evolution facility can also detect pointers to deleted objects and incorrectly typed pointers.

Access control

ObjectStore provides two general approaches to database access control. With one approach, you can set read and write permissions for various categories of users, at various granularities. With the other approach, you can require that applications supply a key in order to access a particular database. ObjectStore also supports Server authentication services.

Concurrency control

The concurrency control scheme employed by ObjectStore is based on transactions with two-phase locking. Locking is automatic and completely transparent to the user. The act of reading (or writing) an object causes a read (write) lock to be acquired for the object, so there is no need to insert any special locking commands into the application code. Locking information is cached on both client and Server, to minimize the need for network communication when the same process performs consecutive transactions on the same data.

ObjectStore also supports multiversion concurrency control, which allows nonblocking database reads. This form of concurrency control is implemented using a technique of delaying propagation of data from the Server log.

Fault tolerance

ObjectStore's fault tolerance is based on transactions and logging. Fault tolerance endows transactions with a number of important properties. Either all of a transaction's changes to persistent memory are made successfully, or none are made at all. If a failure occurs in the middle of a transaction, none of its database updates is made. In addition, a transaction is not considered to have completed successfully until all its changes are recorded safely on stable storage. Once a transaction commits, failures such as server crashes or network failures cannot erase the transaction's changes.

Dynamic schema access

ObjectStore's metaobject protocol supports access to ObjectStore schema information, stored in the form of objects that represent C++ types. These objects are actually instances of ObjectStore *metatypes*, so called because they are types whose instances represent types. Schema information is also represented with the help of various auxiliary classes that are not in the metatype hierarchy, such as ones whose instances represent data members and member functions. The metaobject protocol supports run-time read access to ObjectStore schemas, as well as dynamic type creation and schema modification.

Schema evolution

The term *schema evolution* refers to the changes undergone by a database's schema during the course of the database's existence. It refers especially to schema changes that potentially require changing the representation of objects already stored in the database.

Without the schema evolution facility, a database schema can be changed only by adding new classes to it. Redefinition of a class already contained in the schema, except in ways that do not affect the layout the class defines for its instances, is not allowed. (Adding a nonstatic data member, for example, changes instance layout, but adding a nonvirtual member function does not.)

The schema evolution facility, however, allows arbitrary redefinition of the classes in a database's schema — even if instances of the redefined classes already exist. Invoking schema evolution directs ObjectStore to modify a database's schema, and change the representation of any existing instances in the database to conform to the new class definitions. If desired, these representation changes can be directed by user-supplied routines.

Data compaction

ObjectStore databases consist of segments containing persistent data. As persistent objects are allocated and deallocated in a segment, internal fragmentation in the segment can increase because of the presence of holes produced by deallocation. Of course, the ObjectStore allocation algorithms recycle deleted storage when objects are allocated, but there might nevertheless be a need to compact persistent data by squeezing out the deleted space. Such compaction frees persistent storage space so that it can be used by other segments.

Chapter 2 Class Library

This chapter describes the system-supplied C++ classes, whose members and enumerators provide an interface to database features. The classes are listed alphabetically by class name. Within the entry for each class, the class's members, as well as enumerators defined within the class's scope, are listed alphabetically.

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.

On AIX, Sun, and HP platforms, you can use EUC or SJIS encoding for strings passed to **char*** formal parameters.

ObjectStore classes

objectstore	11
os_access_modifier	44
os_address_space_marker	
os_anonymous_indirect_type	49
os_app_schema	50
os_application_schema_info	
os_array_type	52
os_base_class	54
os_class_type	57
os_comp_schema	70
os_database	71
os_database_reference	101
os_database_root	103

os_database_schema	105
os_dbutil	106
os_DLL_finder	128
os_DLL_schema_info	130
os_dynamic_extent	132
os_enum_type	135
os_enumerator_literal	137
os_evolve_subtype_fun_binding	138
os_failover_server	139
os_field_member_variable	141
os_function_type	142
os_indirect_type	144
os_instantiated_class_type	145
os_integral_type	146
os_literal	148
os_literal_template_actual_arg	151
os_lock_timeout_exception	152
os_member	154
os_member_function	159
os_member_namespace	164
os_member_type	165
os_member_variable	166
os_mop	170
os_named_indirect_type	174
os_namespace	175
os_notification	176
os_object_cluster	186
os_object_cursor	189
os_pathname_and_segment_number	191
os_pointer_literal	192
os_pointer_to_member_type	193
os_pointer_type	194
os_pragma	195
os_pvar	196

os_rawfs_entry 199
os_real_type 202
os_Reference 203
os_reference 209
os_Reference_local 215
os_reference_local 220
os_Reference_protected 224
os_reference_protected 231
os_Reference_protected_local 237
os_reference_protected_local 242
os_Reference_this_DB 247
os_reference_this_DB 252
os_Reference_transient 257
os_reference_transient 262
os_reference_type 266
os_relationship_member_variable 267
os_retain_address
os_schema 270
os_schema_evolution 272
os_schema_handle 289
os_schema_info 293
os_schema_install_options 294
os_segment 295
os_segment_access 307
os_server 311
os_str_conv
os_subscription 318
os_template 321
os_template_actual_arg 323
os_template_formal_arg 325
os_template_instantiation 327
os_template_type_formal 329
os_template_value_formal 330
os_transaction 331

os_transaction_hook 339
os_transformer_binding 341
os_type 342
os_type_template 354
os_type_template_actual_arg 355
os_type_type 356
os_typed_pointer_void 357
os_typed_pointer_void 357
os_void_type 363
os_with_mapped 364

objectstore

This class provides static members related to persistence, performance tuning, and performance monitoring.

Required header files

All ObjectStore programs must include the header file <**ostore/ostore.hh**>.

objectstore::abort_in_progress()

```
static os_boolean abort_in_progress();
```

Returns nonzero if an abort is in progress; returns **0** otherwise.

objectstore::acquire_lock()

```
enum os_lock_type { os_read_lock, os_write_lock, os_no_lock } ;

static os_lock_timeout_exception *acquire_lock(
    void *addr,
    os_lock_type lock_type,
    os_int32 milliseconds,
    os_unsigned_int32 bytes_to_lock = 1
);
```

Attempts to acquire a lock of the type specified by **lock_type** (either **os_read_lock** or **os_write_lock**) on the page(s) containing the memory starting at **addr** and spanning **bytes_to_lock** bytes.

If the lock is successfully acquired, **0** is returned.

Specifying a **–1** value for the **milliseconds** arguments means that **acquire_lock** uses the segment's current **readlock_timeout** or **writelock_timeout** value depending on the type of lock being acquired.

If the caller wants an infinite timeout and the segment's timeout values are not **–1**, the caller could pass a very large value for the timeout (to be effectively infinite). It could also use one of the **objectstore::set_readlock_timeout** or **objectstore::set_writelock_ timeout** entrypoints to set the default to **–1** temporarily.

Specifying a **0** value for the **milliseconds** arguments means that the attempt to acquire the lock will not wait at all if any concurrency conflict is encountered.

After an attempt to acquire a lock, if the time specified by **milliseconds** elapses without the lock's becoming available, an **os_**

lock_timeout_exception* is returned. The timeout is rounded up to the nearest whole number of seconds. The **os_lock_timeout_exception** contains information on the circumstances preventing lock acquisition. It is the caller's responsibility to delete the **os_lock_timeout_exception** object when no longer needed.

If the attempt causes err_deadlock to be signaled in the current process, the transaction is aborted regardless of the value of the specified timeout.

```
static os_lock_timeout_exception *acquire_lock(
    os_database *db,
    os_lock_type access,
    os_int32 milliseconds
);
```

Attempts to acquire a lock of the type specified by **lock_type** (either **os_read_lock** or **os_write_lock**) on the database specified by **db**. Locking a database is equivalent to acquiring locks on all the pages (and segments) of the database. So, for example, acquiring a read lock on a database is equivalent to acquiring read locks on all the pages of the database. Acquiring a lock on a database does not preclude clients from requesting separate locks on individual pages of the database.

If the lock is successfully acquired, **0** is returned.

Specifying a **–1** value for the **milliseconds** arguments means that **acquire_lock** uses the segment's current **readlock_timeout** or **writelock_timeout** value depending on the type of lock being acquired.

If the caller wants an infinite timeout and the segment's timeout values are not **–1**, the caller could pass a very large value for the timeout (to be effectively infinite). It could also use one of the **objectstore::set_readlock_timeout** or **objectstore::set_writelock_timeout** entrypoints to set the default to **–1** temporarily.

Specifying a **0** value for the **milliseconds** arguments means that the attempt to acquire the lock will not wait at all if any concurrency conflict is encountered.

After an attempt to acquire a lock, if the time specified by **milliseconds** elapses without the lock's becoming available, an **os_lock_timeout_exception*** is returned. The timeout is rounded up to the nearest whole number of seconds. The **os_lock_timeout_**

exception contains information on the circumstances preventing lock acquisition. It is the caller's responsibility to delete the **os_lock_timeout_exception** object when it is no longer needed.

If the attempt causes err_deadlock to be signaled in the current process, the transaction is aborted regardless of the value of the specified timeout.

```
static os_lock_timeout_exception *acquire_lock(
    os_segment *seg,
    os_lock_type access,
    os_int32 milliseconds
);
```

Attempts to acquire a lock of the type specified by **lock_type** (either **os_read_lock** or **os_write_lock**) on the segment specified by **seg**. This must be specified in a top-level transaction.

Locking a segment is equivalent to acquiring locks on all the pages of the segment. So, for example, acquiring a read lock on a segment is equivalent to acquiring read locks on all the pages of the segment. Acquiring a lock on a segment does not preclude that A client that has acquired a lock on a segment can also request separate locks on individual pages of the segment.

If the lock is successfully acquired, **0** is returned.

Specifying a **–1** value for the **milliseconds** arguments means that **acquire_lock** uses the segment's current **readlock_timeout** or **writelock_timeout** value depending on the type of lock being acquired.

If the caller wants an infinite timeout and the segment's timeout values are not **–1**, the caller could pass a very large value for the timeout (to be effectively infinite). It could also use one of the **objectstore::set_readlock_timeout** or **objectstore::set_writelock_timeout** entry points to set the default to **–1** temporarily.

Specifying a **0** value for the **milliseconds** arguments means that the attempt to acquire the lock will not wait at all if any concurrency conflict is encountered.

After an attempt to acquire a lock, if the time specified by **milliseconds** elapses without the lock's becoming available, an **os_lock_timeout_exception*** is returned. The timeout is rounded up to the nearest whole number of seconds. The **os_lock_timeout_**

exception contains information on the circumstances preventing lock acquisition. It is the caller's responsibility to delete the **os_lock_timeout_exception** object when it is no longer needed.

If the attempt causes err_deadlock to be signaled in the current process, the transaction is aborted regardless of the value of the specified timeout.

objectstore::add_missing_dispatch_table_handler()

```
typedef void* (*os_missing_dispatch_table_handler_function)
    const char* dispatch_table_identifier, const char*
        dispatch_table_symbol
);
```

```
static void add_missing_dispatch_table_handler(
    os_missing_dispatch_table_handler_function
);
```

Registers the specified **os_missing_dispatch_table_handler_function**. During inbound relocation of a given page, if an object's vtbl slot is not satisfied by any known vtbls, the handler gets called with a string denoting a path to the vtbl slot and a string denoting the platform-dependent linker symbol associated with the vtbl identifier if known.

For example, given

```
    class A { virtual vf1(); };
    class B { virtual vf2(); };
    class C : public A, public B { virtual vf3(); };
```

ObjectStore calls the user's handler function with the string "C@B" if it cannot satisfy the virtual function table slot for the base class subobject B in class C. The **dispatch_table_symbol** is the compiler-specific linker symbol that ObjectStore associates with the dispatch table, or null if the vtbl identifier has no entry in the application schema source file linked in to the application.

The **dispatch_table_symbol** is provided to allow an application to load a library dynamically and look up the symbol. It is also useful for generating a missing vtbl.

objectstore::change_array_length()

```
static void *change_array_length(
    void *array,
    os_unsigned_int32 new_length
);
```

Reallocates the specified persistent array to have the specified number of elements. Returns the address of the reallocated array. The reallocation might or might not be in place. If the value returned is different from the argument, the argument storage has been deleted and is no longer valid. **change_array_length** is similar to the C function **realloc**, but it works for persistent objects only. This function does not execute any constructors or destructors; it simply changes the amount of persistent storage available for the array. In future releases, the behavior of this function might change with regard to when in-place reallocation is performed.

objectstore::compact()

```
static void compact(
    os_char_p *dbs_to_be_compacted,
    os_pathname_and_segment_number_p
        *segments_to_be_compacted = 0,
    os_char_p *dbs_referring_to_compacted_ones = 0,
    os_pathname_and_segment_number_p
        *segments_referring_to_compacted_ones = 0
);
```

Compacts the data in the specified databases and segments, and reorganizes any collections that reference compacted objects. Programs using this function must link with **liboscmp.a**.

dbs_to_be_compacted is a null-terminated array of pointers to **os_char_p** strings identifying the set of databases to be compacted.

os_pathname_and_segment_number_p is a null-terminated array of pointers to **pathname_and_segment_number** objects identifying the segments to be compacted.

dbs_referring_to_compacted_ones is a null-terminated array of pointers to **os_char_p** strings identifying the set of databases containing pointers (or references) to the databases or segments being compacted.

segments_referring_to_compacted_ones is a null-terminated array of pointers to **os_pathname_and_segment_number_p** objects identifying the segments containing pointers (or references) to the databases being compacted.

Either the first or second argument, but not both, can be null; the second argument defaults to null. The set of segments to be compacted is the union of all the data segments in all databases specified by the first argument, plus those segments specified in the optional second argument.

The third and fourth arguments are optional, and if supplied are sets of database pathnames and segment objects containing references to objects in the databases and segments to be compacted. If they are not supplied the compactor assumes that there are no other pointers or references to the segments being compacted.

It is the caller's responsibility to delete the storage associated with the arguments when the function returns.

Compaction-specific invalid arguments will result in the exception err_os_compaction's being signaled.

The **objectstore::compact()** function must be invoked outside any ObjectStore transaction. It is the caller's responsibility to delete the storage associated with the arguments to **objectstore::compact()** upon its return.

If you want to run compaction in a separate process, the application can start up another process that calls the function.

The compactor compacts all C and C++ persistent data, including ObjectStore collections, indexes, and bound queries, and correctly relocates pointers and all forms of ObjectStore references to compacted data. ObjectStore **os_reference_local** references are relocated, assuming that they are relative to the database containing them. The compactor respects ObjectStore clusters, in that compaction will ensure that objects allocated in a particular cluster remain in the cluster, although the cluster itself might move as a result of compaction.

This function operates under the following restrictions:

- *Unions requiring user discriminant functions:* The compactor will not execute union discriminant functions. Therefore, databases containing unions cannot be compacted.
- *Data types requiring user transforms when they move:* The classic example of a data structure that might require user transformation is a hash table that hashes on the offset of an object within a segment. Since compaction modifies these offsets, there is no way such an implicit dependence on the

segment offset can be accounted for by compaction. Of course, transformation of ObjectStore collections is supported in the compactor. Support for invocation of user data transforms will be provided in a future release.

- Since the ObjectStore **retain_persistent_addresses** facility requires that persistent object locations within a segment remain invariant, no client application using this facility and referencing segments to be compacted can run concurrently with the ObjectStore compactor.
- Transient ObjectStore references into a compacted segment become invalid after compaction completes.

ObjectStore supports two file systems for storing databases, and the compactor can run against segments in databases in either file system. In the first and most common case, a single database is stored in a single host system file. The segments in such a database are made up of extents, all of which are allocated in the space provided by the host operating system for the single host file. When there are no free extents left in the host file, and growth of an ObjectStore segment is required, the ObjectStore Server will extend the host file to provide the additional space. The compactor permits holes contained in segments to be compacted to be returned to the allocation pool for the host file, and hence that space can be used by other segments in the same database. However, since operating systems provide no mechanism to free disk space allocated to regions internal to the host file, any such free space will remain inaccessible to other databases stored in other host files.

The ObjectStore rawfs, on the other hand, stores all databases in a single region, either one or more host files or a raw partition. When you are using the raw file system, any space freed by the compaction operation can be reused by any segment in any database stored in the raw file system.

objectstore::discriminant_swap_bytes()

```
static void discriminant_swap_bytes(
    char *address, char *result, os_int32 n_bytes
);
```

When you are accessing, from within a discriminant function, databases created by clients with a different byte ordering, access to a data member whose value occupies more than a single byte must be mediated by **objectstore::discriminant_swap_bytes()**. This function performs byte swapping that ObjectStore normally performs automatically. Use of this function is *only* required within discriminant functions.

The **address** argument is the address of the member whose access is being mediated by this function. The **n_bytes** argument is the number of bytes to swap; possible values are 2, 4, and 8. The result argument points to memory allocated by the user to hold the correctly byte-swapped result; the allocated space should contain **n_bytes** bytes.

objectstore::embedded_server_available()

static os_boolean embedded_server_available();

ObjectStore / Single

Returns nonzero if the ObjectStore / Single version of **libos** is available in the application; returns **0** otherwise.

Functions that report on embedded Servers and on network Servers are mutually exclusive. That is, **objectstore::embedded_ server_available** and **objectstore::network_servers_available** cannot both return true in the same application.

objectstore::enable_damaged_dope_repair()

static void enable_damaged_dope_repair(os_boolean);

Component Schema

Enables or disables automatic repair of incorrect compiler dope for an object while loading a DLL schema. The default is **false.**

Damaged dope repair repairs compiler dope damage by regenerating compiler dope in all cached user data pages of affected databases.

You can query whether dope damage repair is enabled or disabled by calling the function **objectstore::is_damaged_dope_ repair_enabled().**

If dope repair is not enabled, dope damage while loading a DLL schema throws an err_transient_dope_damaged exception.

If dope damage is enabled, dope damage while loading a DLL schema causes ObjectStore to examine each cached user (that is, nonschema) data page, of each segment that contains any transient dope, of each affected database. If the page is currently accessible, ObjectStore immediately regenerates its transient dope (through relocation), otherwise ObjectStore marks the page and its transient dope is regenerated the next time the page is touched.

Compiler Dope

Compiler dope is additional information added to the run-time layout of an object by the compiler,in addition to the nonstatic data members of the object. The correct compiler dope for an object can change as a result of loading or unloading a DLL schema, for example because the compiler dope can point to a virtual function implementation contained in a DLL that is being loaded or unloaded.

Note that an object can suffer dope damage when the implementation of its class changes, even if the object is never used by the program. This is because ObjectStore brings entire pages of databases into memory at a time. If an object is on the same page as another object that is being used, then the first object is also being used as far as dope damage is concerned.

When the combined program schema is rebuilt because a DLL schema has been unloaded, compiler dope in cached persistent objects always needs to be repaired (assuming that there could have been compiler dope pointing to a DLL that was unloaded). This repair takes place regardless of the setting of **objectstore::enable_damaged_dope_repair_enabled()**. The repair procedure is the same as described previously.

objectstore::enable_DLL_schema()

Component Schema

static void enable_DLL_schema(os_boolean);

Enables or disables the component schema feature. The default is **true** for Windows and Solaris platforms.

Use **objectstore::is_DLL_schema_enabled** to query whether DLL schema support is enabled.

objectstore::find_DLL_schema()

```
static os_schema_handle*  find_DLL_schema(
    const char* DLL_identifier,
    os_boolean load_if_not_loaded,
    os_boolean error_if_not_found
);
```

Returns a pointer to the DLL schema handle for the DLL identified by the first argument in the following cases:

- The DLL schema is loaded and not queued for unload.
- The DLL schema is already queued for loading.

Otherwise, if **load_if_not_loaded** is true, calls **objectstore::load_DLL()** with the first and third arguments. If **objectstore::load_DLL()** returns a value other than **os_null_DLL_handle**, this function looks for the DLL schema again.

Note that **objectstore::load_DLL** throws the exception err_DLL_not_loaded if the DLL cannot be found or loaded or the **DLL_identifier** cannot be understood and **error_if_not_found** is true.

If the DLL schema is still not found, and if **error_if_not_found** is true, this function throws an err_schema_not_found exception, otherwise returns a null pointer. Note that if **load_if_not_loaded** and **error_if_not_found** are both true, the exception thrown is err_DLL_not_loaded.

objectstore::get_address_space_generation_number()

os_unsigned_int32 get_address_space_generation_number()

Returns an unsigned integer that is incremented by the client whenever it releases any address space. Its primary purpose is to support pointer caching, such as that used by ObjectStore collections in several circumstances.

A transient cache of persistent pointers should be considered invalid whenever the value of **objectstore::get_address_space_generation_number()** increases. The **objectstore::get_address_space_generation_number()** function simply returns the value read from a variable, and so is fast enough to be called whenever a pointer cache is examined.

objectstore::get_all_servers()

```
static void get_all_servers(
    os_int32 max_servers,
    os_server_p *servers,
    os_int32& n_servers
);
```

Provides access to instances of **os_server** that represent all the ObjectStore Servers known to the current process. The **os_server_p*** is an array of pointers to **os_server** objects. This array must be allocated by the user. The function **objectstore::get_n_servers()**

can be used to determine how large an array to allocate. **max_servers** is specified by the user, and is the maximum number of elements the array is to have. **n_servers** refers to the actual number of elements in the array.

objectstore::get_application_schema_pathname()

```
static const char *get_application_schema_pathname();
```

Returns the pathname of the application schema database.

objectstore::get_as_intervals()

```
static void get_as_intervals(
    os_as_interval_p *persist,
    os_int32& n_persist_intervals,
    os_as_interval_p *other,
    os_int32& n_other_intervals
);
```

This function tells the caller all the ranges of virtual address space that ObjectStore is using, other than ordinary code, text, and heap space. It is primarily intended for users who are trying to integrate ObjectStore and other complex subsystems into the same application.

The class **os_as_interval** has the following public data members:

```
char *start;
os_unsigned_int32 size;
```

The ranges are returned in two sets. The first set of ranges are those used for mapping in persistent objects; the second set is any other ranges of address space that ObjectStore uses.

objectstore::get_auto_open_mode()

```
static void get_auto_open_mode(
    auto_open_mode_enum &mode,
    os_fetch_policy &fp,
    os_int32 &bytes);
```

Returns the value of the current settings for process-specific values for **os_auto_open_mode** and **os_fetch_policy**, and the number of bytes used by the fetch policy.

See also **objectstore::set_auto_open_mode()** on page 33 and **os_database::set_fetch_policy()** on page 95.

objectstore::get_autoload_DLLs_function()

static os_autoload_DLLs_function get_autoload_DLLs_function();

Component Schema

Gets the hook function that is called when a database is put in use and its required DLL set is not empty.

objectstore::get_cache_file()

static char *get_cache_file();

ObjectStore / Single

Returns the name of a cache file previously set with **objectstore::set_cache_file()**; returns **0** if no cache file was specified. This API is only meaningful for ObjectStore / Single applications.

It is the caller's responsibility to deallocate the returned string when it is no longer needed.

objectstore::get_cache_size()

static os_unsigned_int32 get_cache_size();

Returns the current size in bytes of the client cache.

objectstore::get_check_illegal_pointers()

static os_boolean get_check_illegal_pointers();

Returns nonzero (that is, true) if the current process enables **default_check_illegal_pointers** mode for newly created databases; returns **0** (that is, false) otherwise. See **objectstore::set_check_ illegal_pointers()** on page 35.

objectstore::get_incremental_schema_installation()

static os_boolean get_incremental_schema_installation();

Returns nonzero (that is, true) if incremental schema installation is currently enabled; returns **0** (that is, false) if batch schema installation is enabled. See **objectstore::set_incremental_schema_ installation()** on page 37.

objectstore::get_locator_file()

static char* get_locator_file() const;

Returns a string representing the locator file. If the first character of the string is a white-space character or #, the string is the contents of the file rather than a file name.

The caller should delete the returned value.

objectstore::get_lock_status()

static os_int32 get_lock_status(void *address);

Returns one of the following enumerators: **os_read_lock**, **os_write_lock**, **os_no_lock**, indicating the current lock status of the data at the specified address.

objectstore::get_log_file()

static char *get_log_file();

ObjectStore / Single

Returns the name of the Server log file, if set; returns **0** otherwise. This function is only meaningful for ObjectStore / Single applications.

It is the caller's responsibility to deallocate the returned string when it is no longer needed.

objectstore::get_n_servers()

static os_int32 get_n_servers();

Returns the number of ObjectStore Servers to which the current process is connected.

objectstore::get_null_illegal_pointers()

static os_boolean get_null_illegal_pointers();

Returns nonzero (that is, true) if the current process enables **default_null_illegal_pointers** mode for newly created databases; returns **0** (that is, false) otherwise. See **objectstore::set_null_illegal_pointers()** on page 38.

objectstore::get_object_range()

```
static void get_object_range(
    void const *address,
    void *&base_address,
    os_unsigned_int32 &size
);
```

Tells you where a persistent object starts and how large it is. **address** should be a pointer to a persistent object, or into the middle of a persistent object. **base_address** is set to the address of the beginning of the object, and **size** is set to the size of the object in bytes. Arrays are considered to be one object; if **address** is the address of one of the array elements, **base_address** is set to the address of the beginning of the array. If **address** does not point to a persistent object, **base_address** and **size** are both set to **0**.

objectstore::get_opt_cache_lock_mode()

```
static os_boolean get_opt_cache_lock_mode();
```

Returns nonzero if **opt_cache_lock_mode** is on for the current process; returns **0** otherwise. See **objectstore::set_opt_cache_lock_mode()** on page 39.

objectstore::get_page_size()

```
static os_unsigned_int32 get_page_size();
```

Returns the page size for the architecture on which ObjectStore is running.

objectstore::get_pointer_numbers()

```
static void get_pointer_numbers(
    const void *address,
    os_unsigned_int32 &number_1,
    os_unsigned_int32 &number_2,
    os_unsigned_int32 &number_3
);
```

Provides a way for an application to generate a hash code based on object identity. Applications should *not* generate hash codes by casting a pointer to the object into a number, since the address of an object can change from transaction to transaction. Based on the **address** supplied by the caller, the function returns **number_1**, **number_2**, and **number_3**.

Use **number_1** and **number_3** only; ignore **number_2**.

These values will always be the same for a given object, no matter what address it happens to be mapped to at a particular time. Moreover, no two objects will have the same values.

objectstore::get_readlock_timeout()

```
static os_int32 get_readlock_timeout();
```

Returns the time in milliseconds for which the current process will wait to acquire a read lock. A value of **–1** indicates that the process will wait forever if necessary.

objectstore::get_retain_persistent_addresses()

```
static os_boolean get_retain_persistent_addresses();
```

Returns nonzero (that is, true) if the current process is in **retain_ persistent_addresses** mode; returns **0** otherwise. See **objectstore::retain_persistent_addresses()** on page 31.

objectstore::get_simple_auth_ui()

```
static void get_simple_auth_ui(
    void(*&handler)
        (os_void_p, os_char const_p, os_char_p, os_int32,
            os_char_p, os_int32),
    void *&data
);
```

Retrieves the authentication handler information stored by **objectstore::set_simple_auth_ui()**. **handler** is the function that will be called to determine the user and password information needed for authentication. **data** is the user-supplied value that will be passed to the handler function.

objectstore::get_thread_locking()

```
static os_boolean get_thread_locking();
```

If nonzero is returned, ObjectStore thread locking is enabled; if **0** is returned, ObjectStore thread locking is disabled. See **objectstore::set_thread_locking()** on page 40.

objectstore::get_transient_delete_function()

```
static void (*)(void*) get_transient_delete_function();
```

Returns a pointer to the transient delete function last specified by the current process. Returns **0** if there is no current transient delete function. See **objectstore::set_transient_delete_function()** on page 41.

objectstore::get_writelock_timeout()

```
static os_int32 get_writelock_timeout();
```

Returns the time in milliseconds for which the current process will wait to acquire a write lock. A value of **–1** indicates that the process will wait forever if necessary.

objectstore::hidden_write()

```
static void hidden_write(
    char *src_address,
    char *dst_address,
    os_unsigned_int32 len
);
```

src_address points to a transient address, and **dst_address** points to a persistent address. This function writes **len** bytes pointed to by **src_address** into the **dst_address**, without write-locking the object at **dst_address** and without marking the object as modified. This function should only be used from within an access hook (see **os_database::set_access_hooks()** on page 91), and it should only write to locations inside the object for which this access hook is invoked. Typically, the access hook should only use this function to write to locations in the range of addresses that are being made accessible or inaccessible. The persistent value of any location that is used as the target of an **objectstore::hidden_write()** should never be examined by any program.

objectstore::ignore_locator_file()

```
static void ignore_locator_file(os_boolean);
```

Passing a nonzero value ensures that no locator file is associated with the application, regardless of the setting of **OS_LOCATOR_FILE** or calls to **set_locator_file()**. This function is, however, subordinate to the client environment variable **OS_IGNORE_LOCATOR_FILE**. Passing **0** undoes the effect of the previous call to this function.

objectstore::initialize()

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

Must be executed in a process before any use of ObjectStore functionality is made, with the following exceptions:

- **objectstore::propagate_log()** (ObjectStore / Single only)
- **objectstore::set_application_schema_pathname()**
- **objectstore::set_cache_file()** (ObjectStore / Single only)
- **objectstore::set_cache_size()**
- **objectstore::set_client_name()**

These functions *must* be called *before* **objectstore::initialize()**.

A process can execute **initialize()** more than once; after the first execution, calling this function has no effect.

static void initialize(os_boolean force_full_initialization);

Can be used instead of the no-argument overloading of **initialize()**. If **force_full_initialize** is nonzero, all ObjectStore initialization procedures are performed immediately. If **force_full_initialize** is **0**, this function defers some initialization until needed (for example, until a database is first opened). If **force_full_initialize** is **0**, this function behaves just like the no-argument overloading of **initialize()**. Applications that integrate with third-party software might need to force full initialization.

objectstore::is_damaged_dope_repair_enabled()

static os_boolean is_damaged_dope_repair_enabled();

Returns a boolean value indicating whether dope damage repair during DLL schema loading is enabled. The initial state is false. See **objectstore::enable_damaged_dope_repair()**18 for details.

objectstore::is_DLL_schema_enabled()

os_boolean is_DLL_schema_enabled();

Returns whether the DLL schema feature is enabled. The initial state is true on most platforms including Windows and Solaris.

objectstore::is_lock_contention()

static os_boolean is_lock_contention();

Returns nonzero if a server involved in the current transaction has experienced contention for some persistent memory that the calling application is using. Returns **0** otherwise.

This function can be used in conjunction with MVCC to help determine whether to start a new transaction in order to make available more up-to-date data. If your application has a database open for MVCC, and during the current transaction another application has write-locked a page read by your application, **is_lock_contention()** returns nonzero.

If this function is not called from within a transaction, err_trans is signaled.

Note that this function is advisory — it does not have to be called and its return value can be ignored without jeopardizing in any way the correctness of ObjectStore's behavior.

objectstore::is_persistent()

static os_boolean is_persistent(void const *address);

Returns nonzero (true) if the specified address points to persistent memory, and returns **0** (false) otherwise. A pointer to any part of a persistently allocated object (including, for example, a pointer to a data member of such an object) is considered to point to persistent memory. Similarly, a pointer to any part of a transiently allocated object is considered to point to transient memory.

objectstore::load_DLL()

```
static os_DLL_handle load_DLL(
    const char* DLL_identifier,
    os_boolean error_if_not_found = true
);
```

Component Schema

Loads the DLL identified by the **DLL_identifier** and returns an **os_DLL_handle** to it after running its initialization function.

If the DLL cannot be found or the **DLL_identifier** cannot be understood, and **error_if_not_found** is false, this function returns **os_null_DLL_handle**.

If the DLL cannot be found or the **DLL_identifier** cannot be understood and **error_if_not_found** is true, this function throws the exception err_DLL_not_loaded.

If currently in a transaction, aborting the transaction does not roll back **load_DLL()**. The effects of trying to load a DLL that is already loaded has platform-dependent effects.

A DLL can have multiple identifiers, each of which works only on a subset of platforms. The automatic DLL loading mechanism always sets **error_if_not_found** to false and tries all the identifiers.

An error while trying to load the DLL, other then failure to find the DLL, will throw an exception regardless of the setting of **error_if_not_found**. This could occur if an error occurs while executing the DLL's initialization code, for example.

Unix platform note

There is a bug in most versions of Unix that will cause some such errors to look like "DLL not found," and thus be subject to **error_ if_not_found**.

objectstore::lock_as_used

This enumerator is a possible argument to **os_segment::set_lock_ whole_segment()**. It specifies the default behavior, which is initially to lock just the page faulted on when pages are cached. See **os_segment::set_lock_whole_segment()** on page 303.

objectstore::lock_segment_read

This enumerator serves as a possible argument to **os_ segment::set_lock_whole_segment()**. A value of **lock_segment_ read** causes pages in the segment to be read-locked when cached in response to attempted access by the client. Subsequently, upgrading to read / write locks occurs on a page-by-page basis, as needed.

objectstore::lock_segment_write

This enumerator serves as a possible argument to **os_ segment::set_lock_whole_segment()**. A value of **lock_segment_ write** causes pages to be write-locked when cached in response to attempted read or write access by the client. In this case, the Server assumes from the start that write access to the entire segment is desired.

objectstore::lookup_DLL_symbol()

```
static void* objectstore::lookup_DLL_symbol(
    os_DLL_handle h,
    const char* symbol
);
```

Looks up the symbolically named entry point in the DLL identified by the handle and returns its address. If the DLL does not export a symbol equal to the argument, an err_DLL_symbol_ not_found exception is thrown.

objectstore::network_servers_available()

```
static os_boolean network_servers_available();
```

ObjectStore / Single

For use with ObjectStore / Single applications, returns nonzero if the conventional, networked version of ObjectStore's **libos** is available in the application; returns **0** otherwise.

Functions that report on embedded Servers and on network Servers are mutually exclusive. That is, **objectstore::network_servers_available** and **objectstore::embedded_server_available** cannot both return true in the same application.

objectstore::propagate_log()

static void propagate_log(const char *log_path);

ObjectStore / Single

For use with ObjectStore / Single applications. Causes all committed data in the specified Server log to be propagated to the affected databases. Unless errors occur, the log is removed during execution of this call.

An exception is raised (err_not_supported) if this entry point is called from within a full ObjectStore (networked) application.

When used, **objectstore::propagate_log** must be called before **objectstore::initialize**. Most ObjectStore / Single applications will not need to use this entry point since propagation of the application's *own* log file, that is, the one specified by **objectstore::set_log_file**, happens automatically at initialization.

objectstore::release_maintenance()

static os_unsigned_int32 release_maintenance();

Returns the number following the second dot (.) in the number of the release of ObjectStore in use by the current application. For example, for Release 5.0.1, this function would return **1**.

objectstore::release_major()

static os_unsigned_int32 release_major();

Returns the number preceding the first dot (.) in the number of the release of ObjectStore in use by the current application. For example, for Release 5.0, this function would return **5**.

objectstore::release_minor()

static os_unsigned_int32 release_minor();

Returns the number following the first dot (**.**) in the number of the release of ObjectStore in use by the current application. For example, for Release 5.0.1, this function would return **0**.

objectstore::release_name()

static const char *release_name();

Returns a string naming the release of ObjectStore in use, for example, "**ObjectStore 5.0**".

objectstore::release_persistent_addresses()

static void release_persistent_addresses();

Globally disables retaining the validity of persistent addresses across transaction boundaries. Used in conjunction with **objectstore::retain_persistent_addresses()**.

This function is callable within top-level transactions as well as outside of a transaction.

objectstore::retain_persistent_addresses()

static void retain_persistent_addresses();

Globally enables retaining the validity of persistent addresses across transaction boundaries. Must be called outside any transaction. Once executed within a given process, pointers to persistent memory remain valid even after the transaction in which they were retrieved from the database. This is true until the end of the process, or until **objectstore::release_persistent_ addresses()** is called.

objectstore::return_all_pages()

static os_unsigned_int32 return_all_pages();

Clears the client cache.

objectstore::return_memory()

```
static os_unsigned_int32 return_memory(
    void *address,
    os_unsigned_int32 length,
    os_boolean evict_now
);
```

Gives the programmer control over cache replacement. The first two arguments designate a region of persistent memory; **address** is the beginning of the range and **length** is the length of the range in bytes. The function tells ObjectStore that this region of persistent memory is unlikely to be used again in the near future. If **evict_now** is nonzero (true), the pages are evicted from the cache immediately. If **evict_now** is **0** (false), the pages are not immediately evicted, but they are given highest priority for eviction (that is, they are treated as if they are the least recently used cache pages).

objectstore::set_always_ignore_illegal_pointers()

static void set_always_ignore_illegal_pointers(os_boolean);

By default, ObjectStore signals an exception when it detects an illegal pointer (a pointer from persistent memory to transient memory or a cross-database pointer from a segment that is not in **allow_external_pointers** mode). Supplying a nonzero value specifies that illegal pointers should always be ignored by ObjectStore during the current process, provided the process is not in **always_null_illegal_pointers** mode. This includes illegal pointers detected during database reads as well as database writes.

objectstore::set_always_null_illegal_pointers()

static void set_always_null_illegal_pointers(os_boolean);

Supplying a nonzero value specifies that illegal pointers should always be set to **0** when detected by ObjectStore during the current process. This includes illegal pointers detected during database reads as well as database writes.

objectstore::set_application_schema_pathname()

static void set_application_schema_pathname(const char *path);

Specifies the location of the application schema database. This function must be called before **objectstore::initialize()**.

objectstore::set_autoload_DLLs_function()

```
static os_autoload_DLLs_function set_autoload_DLLs_function(
    os_autoload_DLLs_function fcn
);
```

Component Schema

Controls whether DLLs are loaded automatically by setting a hook function that is called when a database is put in use and its required DLL set is not empty. Calling this function returns the previously set hook function.

You can set the hook function to a function that does nothing if you need to disable automatic loading of DLLs.

The default initial value of the hook function works as follows:

1 Call **os_database::get_required_DLL_identifiers**.
2 For each **DLL_identifier**, call **objectstore::find_DLL_schema** with arguments of the DLL identifier, true, and false, and ignore the result.

There is caching to avoid calling the hook function when a database is being put in use for the second or later time in a process, the database's required DLL set has not grown, and the process has not unloaded any DLLs.

objectstore::set_auto_open_mode()

```
enum auto_open_mode_enum
{auto_open_read_only, auto_open_mvcc, auto_open_update,
auto_open_disable};
```

```
static void set_auto_open_mode(
    auto_open_mode_enum mode = auto_open_update,
    os_fetch_policy fp = os_fetch_page,
    os_int32 bytes = 0);
```

Enables auto-open mode. This mode automatically opens any databases that need to be opened due to traversal of a reference or a cross-database-pointer. Specify the mode in which to open the database with one of the following **enum** values:

auto_open_read_only	Opens the database as read-only.
auto_open_mvcc	Opens the database for multi-version concurrency control. See **os_database::open_mvcc()** on page 89
auto_open_update	Opens the database for updates.
auto_open_disable	Disables auto-open mode.

If a database is already open, a nested open is not done on that database. If auto-open mode is disabled, the error err_database_not_open is signaled upon an attempt to do an auto open.

The fetch policy for auto-opened databases can also be set using this interface. See **os_database::set_fetch_policy()** for a discussion of the use of fetch policies.

Warning

Exercise extra caution if you have several databases open for MVCC at once. In particular, be aware that the databases will not necessarily be consistent with each other. Unless you are very careful, this could lead to unexpected results.

objectstore::set_cache_file()

```
static void set_cache_file(
    const char *cache_path,
    os_boolean pre_allocate = 1
);
```

ObjectStore / Single

Names a file to be used as the ObjectStore / Single cache. This entry point has no effect if called in a full ObjectStore (networked) application.

If the **pre_allocate** argument is nonzero (the default), the cache file is explicitly filled with zeros when it is opened. (See **objectstore::set_cache_size()**). Preallocation slows down start-up, but protects against obscure failures of **mmap** at critical times if the file system runs out of space.

If the **pre_allocate** argument is **0** (not the default) and an out-of-disk-space condition occurs when ObjectStore is trying to use a page in the cache file, the reported error is obscure (likely to be err_internal), and the diagnostic message does not say anything about disk space. The error message is most likely to complain about problems with **mmap**, page protections, or possibly page locks.

The call must precede **objectstore::initialize**. It takes precedence over the environment variable **OS_CACHE_FILE**. Be aware that if the file already exists, it is overwritten.

The cache file is not removed when the application ends. Normally, users should do so in the interest of conserving disk space.

Note that cache files can be reused but cannot be shared. An attempt to start an ObjectStore / Single application with a cache file that is already being used by another ObjectStore / Single application results in an error.

objectstore::set_cache_size()

static void set_cache_size(os_unsigned_int32 new_cache_size);

Sets the size of the client cache in bytes. The actual size is rounded down to the nearest whole number of pages. **objectstore::set_ cache_size** must be called before **objectstore::initialize()**. Affects performance only.

objectstore::set_check_illegal_pointers()

static void set_check_illegal_pointers(os_boolean);

If the argument is 1, this directs ObjectStore to create new databases in **default_check_illegal_pointers** mode. It also enables **check_illegal_pointers** mode for each database currently retrieved by the current process. See **os_segment::set_check_illegal_ pointers()** on page 301 and **os_database::set_default_check_ illegal_pointers()** on page 94.

objectstore::set_client_name()

static void set_client_name(char *new_name);

Sets the name of the program that is running. Calling **objectstore::set_client_name("program_name")** during program initialization makes some of the output of the ObjectStore administrative / debugging commands (such as the **-d** option to the Server and the **ossvrstat** command) easier to understand. Must be called before **objectstore::initialize()**.

objectstore::set_commseg_size()

static void set_commseg_size(os_unsigned_int32 bytes);

Sets the size of the *commseg* in bytes. The actual size is rounded to the nearest whole number of pages. The commseg is a preallocated region on each ObjectStore client. It holds data used internally by ObjectStore, including cache indexing information and data that describes databases and segments used by the client.

The space requirements for the commseg are roughly as follows:

- 2000 bytes constant overhead
- 84 bytes for each page in the client cache
- 840 bytes for each segment created or used

- A small number of bytes for each Server and each database used

So, for example, with 15,000 segments and an 8M cache (which is 2048 4K pages), roughly 12774032 (12.7 MB) of commseg is required.

objectstore::set_current_schema_key()

```
static void set_current_schema_key(
    os_unsigned_int32 key_low,
    os_unsigned_int32 key_high
);
```

Sets or unsets the schema key of the current application. Call this function only after calling **objectstore::initialize()**. Otherwise, err_schema_key is signaled and ObjectStore issues an error message like the following:

```
<err-0025-0153> The schema key may not be set until after
objectstore::initialize has been called.
```

key_low specifies the first component of the schema key, and **key_high** specifies the second component. If both these arguments are **0**, calling this function causes the application's schema key to be determined as for an application that has not called this function.

If an application has not called this function, its key is determined by the values of the environment variables **OS_SCHEMA_KEY_HIGH** and **OS_SCHEMA_KEY_LOW**. If both the variables are not set, the application has no current schema key.

See Chapter 7, Database Access Control, in *ObjectStore C++ API User Guide*.

objectstore::set_eviction_batch_size()

```
static void set_eviction_batch_size(os_unsigned_int32);
```

Sets the minimum number of bytes, rounded up to the nearest whole number of pages, evicted when ObjectStore needs to make room in the client cache, assuming there are enough candidates in the eviction pool. If you specify **0**, a batch size of one page is used. If you specify a batch size greater than the size of the client cache, the number of pages in the client cache is used. If this function is not called, the batch size is one percent of the cache size, rounded up to the nearest whole number of pages.

objectstore::set_eviction_pool_size()

static void set_eviction_pool_size(os_unsigned_int32);

Sets the number of bytes, rounded up to the nearest whole number of pages, in the pool out of which an eviction batch is taken. If this function is not called, the batch size is 2% of the cache size rounded up to the nearest whole number of pages or 10 pages, whichever is larger.

objectstore::set_handle_transient_faults()

static void set_handle_transient_faults(os_boolean);

UNIX

(Calls to this function are ignored under Windows.) Determines whether dereferencing an illegal pointer (for example, a null pointer) in the current process causes an operating system signal or an ObjectStore exception. If a nonzero value is supplied as argument, an ObjectStore exception results; if **0** is supplied, or if the function has not been invoked, an operating system signal results.

objectstore::set_incremental_schema_installation()

static void set_incremental_schema_installation(os_boolean);

If a nonzero value (true) is supplied as argument, the current application run will perform incremental schema installation on each database it accesses, regardless of the database's mode. In addition, databases subsequently created by the current execution of the application will be in incremental mode, and the schema of the creating application will be installed incrementally. With incremental schema installation, a class is added to a database's schema only when an instance of that class is first allocated in the database. If **0** (false) is supplied as argument, databases subsequently created by the current execution of the application will be in batch mode (the default). With batch mode, whenever an application creates or opens the database, every class in the application's schema is added to the database's schema (if not already present in the database schema).

objectstore::set_locator_file()

static void set_locator_file(const char *file_name);

The argument **file_name** points to the name of the locator file to be used the next time a database is opened. If **0** is supplied, the client environment variable **OS_LOCATOR_FILE** is used to determine the locator file to use. A nonzero argument overrides any setting of **OS_LOCATOR_FILE**. If the specified file does not exist err_locator_misc is signaled. If the first character of the string pointed to by **file_name** is a white-space character or #, the string is assumed to be the contents of a file rather than a file name.

objectstore::set_log_file()

static void set_log_file(const char *log_path);

ObjectStore / Single

Names a file that will be used for the ObjectStore / Single Server log. This entry point has no effect if called in a full ObjectStore (networked) application. It takes precedence over the environment variable **OS_LOG_FILE**. Note the discussion of considerations about this environment variable in *ObjectStore Management*.

If the file already exists, it must be a valid Server log created by an earlier execution of an ObjectStore / Single application. In that case, all committed data in that log is propagated during ObjectStore initialization.

The log file is normally removed at program termination or when **objectstore::shutdown** is called. If errors occur, the log might not be removed. In that event the user should consider the log to contain unpropagated data.

objectstore::set_mapped_communications()

static void set_mapped_communications(os_boolean);

Passing **1** enables mapped communications between client and Server. Passing **0** disables it. It is enabled by default. This function overrides the environment variable **OS_NO_MAPPED**. See *ObjectStore Management*.

objectstore::set_null_illegal_pointers()

static void set_null_illegal_pointers(os_boolean);

If the argument is **1**, this directs ObjectStore to create new databases in **default_null_illegal_pointers** mode. It also enables **null_illegal_pointers** mode for each database currently retrieved by the current process. See **os_segment::set_null_illegal_pointers()**

on page 304 and **os_database::set_default_null_illegal_pointers()** on page 94.

objectstore::set_opt_cache_lock_mode()

static void set_opt_cache_lock_mode(os_boolean);

A nonzero argument turns on **opt_cache_lock_mode** for the current process; a **0** argument turns the mode off.

Turning on this mode will improve performance for applications that perform database writes and expect little or no contention from other processes for access to persistent memory. When this mode is on, the amount of client / server communication required to upgrade locks is reduced. Once a page is cached on the client, the client can subsequently upgrade the page's lock from read to read / write when needed without communicating with the Server. However, the amount of client / server communication required for concurrent processes to obtain locks might increase.

opt_cache_lock_mode is ignored during read-only transactions.

Note that this function sets the mode for the current process only, and does not affect the mode for other processes.

objectstore::set_readlock_timeout()

static void set_readlock_timeout(os_int32);

Sets the time in milliseconds for which the current process will wait to acquire a read lock. The time is rounded up to the nearest whole number of seconds. A value of **–1** indicates that the process should wait forever if necessary. After an attempt to acquire a read lock, if the specified time elapses without the lock's becoming available, an os_lock_timeout exception is signaled. If the attempt causes a deadlock, the transaction is aborted regardless of the value of the specified timeout.

objectstore::set_reserve_as_mode()

static void set_reserve_as_mode(os_boolean new_mode);

See the documentation for the environment variable **OS_ RESERVE_AS** in *ObjectStore Management*.

objectstore::set_simple_auth_ui()

static void set_simple_auth_ui(

```
    void(*)(os_void_p, os_char_const_p, os_char_p,
            os_int32, os_char_p, os_int32),
    void*
);
```

Registers an authentication handler function.

The first argument is a handler function that will be called by ObjectStore when the application first attempts to access a Server that requires Name Password authentication (see *ObjectStore Management*). The function is responsible for providing user name and password information.

The second argument is a data value that will be passed to the handler function when it is called.

The handler function has the following arguments: the first argument is the **void*** argument that was passed to **objectstore::set_simple_auth_ui()**. The second argument is the name of the Server host. The third and fourth arguments are a pointer to and length of a range of memory into which your function should put the user name. The fifth and sixth arguments are the same, for the password.

If no handler function is registered, the application issues a message to **stdout** requesting a user name and password, when first accessing a Server requiring Name Password authentication. By registering a handler function, you can, for example, use a dialog box instead of standard input and output to obtain authentication information from an end user.

objectstore::set_thread_locking()

```
static void set_thread_locking(os_boolean);
```

To enable ObjectStore thread locking explicitly, pass a nonzero value. To disable ObjectStore thread locking, pass **0** to this function. If you disable ObjectStore thread locking while collections thread locking is enabled, collections thread locking remains enabled. You should disable collections thread locking as well.

If your application does not use multiple threads, disable thread locking with this function and **os_collection::set_thread_locking()**.

If your application uses multiple threads, and the synchronization coded in your application allows two threads to be within the

ObjectStore run time at the same time, you need ObjectStore thread locking enabled. See also os_collection::set_thread_locking() in the *ObjectStore Collections C++ API Reference* and **os_transaction::begin()** on page 332.

objectstore::set_transaction_priority()

static void set_transaction_priority(os_unsigned_int16 priority);

Every client has a *transaction priority*. The value is an unsigned number that can range from 0 to 0xffff. The default value is 0x8000 (which is right in the middle). Note that the value 0 is special, as described below.

When two clients deadlock, the transaction priority is used as part of the decision as to which client should be the *victim*, that is, which one should be forced to abort, and possibly restart, its transaction.

When it makes this decision, the first thing the Server does is to compare the transaction priorities of all the participants. If they do not all have equal priority, the ones with the higher priority are not considered as deadlock victims. That is, it looks at the lowest priority number of all the participants, and any participant with a higher priority number is no longer considered as a possible victim.

If there is only one participant left, it is the victim. Otherwise, if there are several participants that all share the same lowest priority number, it chooses a victim in accordance with the Server parameter **Deadlock Victim**. See *ObjectStore Management*.

There is one important special case. If all the participants have priority zero, the Server will victimize *all* the participants! This is not a useful mode of operation for actually running a program, but it can be useful for debugging: you can run several clients under debuggers, have them all set their priorities to zero, and then when a deadlock happens, all of them abort, and you can see what each one of them was doing. You should never use a priority of zero unless you want this special debugging behavior.

objectstore::set_transient_delete_function()

```
static void set_transient_delete_function(
    void (*)(os_void_p)
);
```

Since ObjectStore redefines the global **operator delete()** so that it can take control of deletion of persistent objects, applications cannot provide their own overloaded global **operator delete()**. However, instead of overloading **::operator delete()** to arrange for application-specific transient deallocation processing, applications can register a transient delete function by passing a pointer to the function to **objectstore::set_transient_delete_function()**. The specified function is user defined, and should do what the application's operator delete would have done. ObjectStore continues to provide the definition of **::operator delete()**. When ObjectStore's **::operator delete()** is given a transient pointer, and **set_transient_delete_function()** has been called, it calls the specified transient delete function on the transient pointer.

The initial value of the delete function is **0**, meaning that ObjectStore's **::operator delete()** should ignore zero pointers and call **free()** (the architecture's native storage-freeing function) on the pointer. You can set the value back to **0** if you want to.

objectstore::set_writelock_timeout()

static void set_writelock_timeout(os_int32);

Sets the time in milliseconds for which the current process will wait to acquire a write lock. The time is rounded up to the nearest whole number of seconds. A value of **–1**, the default, indicates that the process should wait forever if necessary. After an attempt to acquire a write lock, if the specified time elapses without the lock's becoming available, an os_lock_timeout_exception exception is signaled. If the attempt causes a deadlock, the transaction is aborted regardless of the value of the specified timeout.

objectstore::shutdown()

static void shutdown();

Conducts an orderly shutdown of ObjectStore. In particular, all open databases are closed to facilitate propagation of committed data from the Server log to the databases.

For ObjectStore / Single, this call attempts to propagate all committed data in the log and then to remove the log. However, if errors occur, the log might not be removed. In that event the user should consider the log to contain unpropagated data.

There should not be a transaction in progress when this entry point is called.

As currently implemented, ObjectStore cannot be restarted after this entry point is called. Object Design recommends that you use **objectstore::shutdown** for both full ObjectStore and ObjectStore / Single applications.

objectstore::unload_DLL()

static void unload_DLL(os_DLL_handle h);

If **h** designates a loaded DLL, unload it. If **h** is **os_null_DLL_handle**, do nothing. Otherwise the results are platform dependent.

If an operating system error occurs while the DLL is being unloaded, an err_DLL_not_unloaded exception can be thrown.

objectstore::which_product()

static os_product_type which_product();

Always returns **objectstore**.

os_access_modifier

class os_access_modifier : public os_member

This class is part of the ObjectStore metaobject protocol, which provides access to ObjectStore schemas. An instance of this class represents the access modification performed by a class on an inherited member. **os_access_modifier** is derived from **os_ member**.

os_access_modifier::create()

static os_access_modifier& create(os_member*);

Creates an **os_access_modifier** that modifies access to the specified member.

os_access_modifier::get_base_member()

const os_member &get_base_member() const;

Returns a reference to the **const** member whose access was modified.

os_member &get_base_member();

Returns a reference to the non**-const** member whose access was modified.

os_access_modifier::set_base_member()

void set_base_member(os_member&);

Updates the member whose access is to be modified.

os_address_space_marker

To the address space marker feature, create an **os_address_space_marker** at some convenient point where address space consumption that is to be undone is about to begin (like the beginning of a query). Later, when **os_address_space_marker::release()** is called, all address space reservations added since that marker was created will be released (subject to the same restrictions mentioned for **objectstore::release_persistent_addresses** — some address space cannot be released during a transaction).

The API allows markers to be nested; that is, several markers can be in effect at the same time. Calling **os_address_space_marker::release()** on an outer marker releases any markers nested within it.

The **os_address_space_marker::retain()** function allows selective release of address space, similar to creating **os_retain_address** objects, but without the requirement for stack allocation (which binds the usage to a lexical scope). Call **os_address_space_marker::retain()** on any pointers that should remain valid across the release boundary, prior to calling **release**. **os_address_space_marker::retain()** can also be passed a marker — in this case, the address space required by the pointer is not released until that marker is released. It is not possible to use the **retain** function on an address to make it be released sooner, by a more nested marker — attempts to do so are ignored.

The **os_address_space_marker::release()** function releases address space added since the creation of the mark, minus any address space retained by calls to the **os_address_space_marker::retain()** function (and any retained by **os_retain_address** objects and **os_pvars**). **release()** can be called on a marker repeatedly, each time releasing the address space accumulated since the previous release (or since the marker was created).

If a marker is deleted and no call to **os_address_space_marker::release()** is made, the marker is removed and the address spaced that it controlled is now controlled by its previous marker. If there is no previous marker, the address space is not governed by any marker and is no longer incrementally releasable.

After the outermost marker is created, no more than 2^{32-2} minus 1 additional markers can be created before the outermost one is deleted. This is true even if some or all of the inner markers are deleted.

Like **objectstore::release_persistent_addresses()**, markers cannot be released within nested transactions.

The implementation of **os_address_space_marker::release()**, besides possibly not freeing as much address space as **objectstore::release_persistent_addresses()**, also does not cool the client cache as much. **os_address_space_marker::release()** must relocate out all pages that were relocated in or modified after the marker was constructed

os_address_space_marker::get_current()

static os_address_space_marker *get_current();

Returns the current address space marker. The current marker is defined as the most recently constructed marker that has not yet been deleted. This function can be used with nested markers.

os_address_space_marker::get_level()

os_unsigned_int32 get_level() const;

The level of a marker is 1 if there was no current marker when it was created. Otherwise, the level is 1 greater than the level of the previously created marker. Use **get_level()** to quickly compare address space markers. Markers with lower levels come before those with higher levels. This function can be used with nested markers.

os_address_space_marker::get_next()

os_address_space_marker *get_next() const;

Returns the next address space marker (or NULL if **this** marker is the last). The next address space marker is the first one that was created after **this** address space marker.

This function can be used with nested markers.

os_address_space_marker::get_previous()

os_address_space_marker *get_previous() const;

Returns the previous address space marker (or NULL if **this** marker is the first). The previous address space marker is the last one that was created before **this** address space marker. This function can be used with nested markers.

os_address_space_marker::of()

static os_address_space_marker *of(void *p);

Returns the address space marker (or NULL) with the highest level that, when released, will release the address space needed for pointer **p**.

os_address_space_marker::os_address_space_marker()

os_address_space_marker();

Creates an **os_address_space_marker**.

os_address_space_marker::release()

void release();

Releases the address space that was added to the PSR since the address space marker was created, or since the last time **os_ address_space_marker::release** was called.

Deleting a marker does not does not release the address space it has marked. Conversely, releasing a marker does not deactivate, or delete, it. This means you can call the **release** function again on the same marker after more address space has been accumulated. **os_address_space_marker::release** does not affect the value of **os_ address_space_marker::get_current()**. This function can be used with nested markers.

os_address_space_marker::retain()

```
static void retain(
    void *p,
    os_address_space_marker *marker = NULL
);
```

Returns the address space marker (or NULL) with the highest level that, when released, releases the address space needed for pointer **p**. Retains space needed by pointer **p** back to some **marker**, or (if the **marker** is null) back past all markers.

os_address_space_marker::~os_address_space_marker()

~os_address_space_marker();

Destructor function.

.

os_anonymous_indirect_type

class os_anonymous_indirect_type : public os_type

This class is part of the ObjectStore metaobject protocol, which provides access to ObjectStore schemas. An instance of this class represents a **const** or **volatile** type. This class is derived from **os_ type**.

os_anonymous_indirect_type::create()

```
static os_anonymous_indirect_type &create(
    os_type *target_type,
);
```

Creates an anonymous indirect type with the specified **target_type** and **name**.

os_anonymous_indirect_type::get_target_type()

```
const os_type &get_target_type() const;
```

Returns the type to which the **const** or **volatile** specifier applies. For example, the type **const int** is represented as an instance of **os_ anonymous_indirect_type** whose target type is an instance of **os_ integral_type**.

os_anonymous_indirect_type::is_const()

```
os_boolean is_const() const;
```

Sets the name of an **os_anonymous_indirect_type** of type **const**.

os_anonymous_indirect_type::is_volatile()

```
os_boolean is_volatile() const;
```

Sets the name of an **os_anonymous_indirect_type** of type **volatile**.

os_anonymous_indirect_type::set_is_const()

```
void set_is_const(os_boolean);
```

Sets the name of an **os_anonymous_indirect_type** of type **const**.

os_anonymous_indirect_type::set_is_volatile()

```
void set_is_volatile(os_boolean);
```

Sets the name of an **os_anonymous_indirect_type** of type **volatile**.

os_app_schema

This class is part of the ObjectStore metaobject protocol, which provides access to ObjectStore schemas. An instance of this class represents an application schema, stored in an application schema database, or a component schema, stored in a component schema database. **os_app_schema** is derived from **os_schema**.

Programs using this class must include **<ostore/ostore.hh>**, followed by **<ostore/coll.hh>** (if used), followed by **<ostore/mop.hh>**.

os_app_schema::get()

static const os_app_schema &get();

Returns the schema of the application making the call. Signals err_no_schema if the schema could not be found.

static const os_app_schema &get(const os_database&);

Returns the schema of the specified database. Signals err_no_schema if the specified database is not an application or component schema database.

os_application_schema_info

Include files

You must include the header file **<ostore/nreloc/schftyps.hh>**.

Provides Information in an application about its application schema, including the pathname of the schema database, rep descriptors, pointers to vtables, etc.

os_array_type

class os_array_type : public os_type

This class is part of the ObjectStore metaobject protocol, which provides access to ObjectStore schemas. An instance of this class represents a C++ array type. This class is derived from **os_type**.

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.

Programs using this class must include **<ostore/ostore.hh>**, followed by **<ostore/coll.hh>** (if used), followed by **<ostore/mop.hh>**.

os_array_type::create()

```
static os_array_type &create(
    os_unsigned_int32 number_of_elements,
    os_type *element_type
);
```

Creates an array type with the specified number of elements and the specified element type.

os_array_type::get_element_type()

const os_type &get_element_type() const;

Returns the type of element contained in instances of the specified array type.

os_type &get_element_type();

Returns the type of element contained in instances of the specified array type.

os_array_type::get_number_of_elements()

os_unsigned_int32 get_number_of_elements() const;

Returns the number of elements associated with the specified array type. If the number is not known, **0** is returned.

os_array_type::set_element_type()

void set_element_type(os_type &);

Specifies the type of element contained in instances of the specified array type.

os_array_type::set_number_of_elements()

void set_number_of_elements(os_unsigned_int32);

Specifies the number of elements associated with the specified array type.

os_base_class

This class is part of the ObjectStore metaobject protocol, which provides access to ObjectStore schemas. An instance of **os_base_class** represents a class from which another class is derived, together with the nature of the derivation (that is, virtual or nonvirtual, and private, public, or protected).

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.

Programs using this class must include **<ostore/ostore.hh>**, followed by **<ostore/coll.hh>** (if used), followed by **<ostore/mop.hh>**.

os_base_class::create()

```
static os_base_class &create(
    os_unsigned_int32 access,
    os_boolean is_virtual,
    os_class_type *associated_class
);
```

Creates an **os_base_class**. The arguments specify the initial values for the attributes **access**, **is_virtual**, and **associated_class**.

os_base_class::get_access()

```
int get_access() const;
```

Returns an enumerator describing the access to the base class members, **os_base_class::Public**, **os_base_class::Private**, or **os_base_class::Protected**.

os_base_class::get_class()

```
const os_class_type &get_class() const;
```

Returns a reference to a **const os_class_type**, the class serving as base class in the derivation represented by the specified **os_base_class** object.

```
os_class_type &get_class();
```

Returns a reference to a non-**const** **os_class_type**, the class serving as base class in the derivation represented by the specified **os_base_class** object.

os_base_class::get_offset()

os_unsigned_int32 get_offset() const;

Returns the offset in bytes to the base class from the immediately enclosing class. For virtual bases, this offset is only meaningful if the base class was obtained by a call to **os_class_type::get_ allocated_virtual_base_classes()**.

os_unsigned_int32 get_offset(const os_class_type&) const;

Returns the offset in bytes to the base class from the specified most derived class. **this** must be a virtual base class.

os_base_class::get_size()

os_unsigned_int32 get_size() const;

Returns the size in bytes of the base class.

os_base_class::get_virtual_base_class_pointer_offset()

os_unsigned_int32 get_virtual_base_class_pointer_offset() const;

Returns the offset of the virtual base class pointer.

os_base_class::is_virtual()

os_boolean is_virtual() const;

Returns **1** if and only if the specified base class is virtual.

os_base_class::Private

This enumerator is a possible return value from **os_base_ class::get_access()**, indicating **private** access.

os_base_class::Protected

This enumerator is a possible return value from **os_base_ class::get_access()**, indicating **protected** access.

os_base_class::Public

This enumerator is a possible return value from **os_base_ class::get_access()**, indicating **public** access.

os_base_class::set_access()

void set_access(os_unsigned_int32);

Specifies an enumerator describing the access to the base class members, **os_base_class::Public**, **os_base_class::Private**, or **os_ base_class::Protected**.

os_base_class::set_class()

void set_class(os_class_type &);

Specifies the class serving as base class in the derivation represented by the specified **os_base_class** object.

os_base_class::set_offset()

void set_offset(os_unsigned_int32);

Sets the offset in bytes of the base class from the immediately enclosing class.

os_base_class::set_virtual_base_class_no_pointer()

void set_virtual_base_class_no_pointer();

Specifies that the base class has no virtual base class pointer.

os_base_class::set_virtual_base_class_pointer_offset()

void set_virtual_base_class_pointer_offset(os_unsigned_int32);

Sets the offset of the virtual base class pointer.

os_base_class::set_virtuals_redefined()

void set_virtuals_redefined(os_boolean);

Specifies whether the base class redefines any virtual functions.

os_base_class::virtual_base_class_has_pointer()

os_boolean virtual_base_class_has_pointer() const;

Returns nonzero if the base class has a virtual base class pointer.

os_base_class::virtuals_redefined()

os_boolean virtuals_redefined() const;

Returns nonzero if the base class redefines any virtual functions.

os_class_type

class os_class_type : public os_type

This class is part of the ObjectStore metaobject protocol, which provides access to ObjectStore schemas. An instance of **os_class_type** represents a C++ class. **os_class_type** is derived from **os_type**.

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.

Programs using this class must include **<ostore/ostore.hh>**, followed by **<ostore/coll.hh>** (if used), followed by **<ostore/mop.hh>**.

os_class_type::Anonymous_union

This enumerator is a possible return value from **os_class_type::kind()**. It indicates an anonymous union type.

os_class_type::Class

This enumerator is a possible return value from **os_class_type::kind()**. It indicates a class declared with the class-key **class**.

os_class_type::create()

static os_class_type &create(const char *name);

Creates a new class with the specified name (which is copied by ObjectStore). It initializes the other attributes of **os_class_type** as follows:

Attribute	*Value*
base_classes	**empty os_List<os_base_class*>**
members	**empty os_List<os_member*>**
defines_virtual_functions	**0**
class_kind	**os_class_type::Class**
defines_get_os_typespec_function	**0**
is_template_class	**0**
is_persistent	**0**

Attribute	*Value*
is_forward_definition	**1**

```
static os_class_type &create(
    const char *name,
    os_List<os_base_class*> &base_classes,
    os_List<os_member*> &members,
    os_boolean defines_virtual_functions
);
```

Creates a new class. The arguments specify the initial values for the attributes **name**, **base_classes**, **members**, and **defines_virtual_functions**. The initial values for the remaining attributes are as follows:

Attribute	*Value*
class_kind	**os_class_type::Class**
defines_get_os_typespec_function	**0**
is_template_class	**0**
is_persistent	**0**
is_forward_definition	**0**

If you create a class with the metaobject protocol, there is no direct way to make it compiler heterogeneous.

os_class_type::declares_get_os_typespec_function()

```
os_boolean declares_get_os_typespec_function() const;
```

Returns nonzero if and only if the specified class declares a **get_os_typespec()** member function.

os_class_type::defines_virtual_functions()

```
os_boolean defines_virtual_functions() const;
```

Returns nonzero if and only if the specified class defines any virtual functions.

os_class_type::find_base_class()

```
const os_base_class *find_base_class(const char *classname)
const;
```

Returns a pointer to the **const os_base_class** whose class has the given name and is a base class of **this**. Either the inheritance is direct or the base class is a virtual base class. If there is no such class, **0** is returned. err_mop_forward_definition is signaled if the specified class is known only through a forward definition.

os_base_class *find_base_class(const char *classname);

Returns a pointer to the non-**const os_base_class** whose class has the given name and is a base class of **this**. Either the inheritance is direct or the base class is a virtual base class. If there is no such class, **0** is returned. err_mop_forward_definition is signaled if the specified class is known only through a forward definition.

os_class_type::find_member_variable()

const os_member_variable *find_member_variable(
const char *name
) const;

Returns a pointer to a **const os_member_variable** representing the data member of **this** with the given name. The member must be defined by **this**, not inherited by it. If there is no such data member, **0** is returned. err_mop_forward_definition is signaled if the specified class is known only through a forward definition.

os_member_variable *find_member_variable(const char *name);

Returns a pointer to a non-**const os_member_variable** representing the data member of **this** with the given name. The member must be defined by **this**, not inherited by it. If there is no such data member, **0** is returned. err_mop_forward_definition is signaled if the specified class is known only through a forward definition.

os_class_type::get_access_of_get_os_typespec_function()

os_member::os_member_access get_access_of_get_os_typespec_function() const;

Returns **os_member::Private**, **os_member::Protected**, or **os_member::Public**.

os_class_type::get_allocated_virtual_base_classes()

os_list get_allocated_virtual_base_classes() const;

Returns a list of pointers to **const os_base_class** objects. Each **os_base_class** object represents a virtual base class from which the specified class inherits (that is, whose storage is allocated as part of the specified class). The order of list elements is not significant.

err_mop_forward_definition is signaled if the specified class is known only through a forward definition.

os_list get_allocated_virtual_base_classes();

Returns a list of pointers to non-**const** **os_base_class** objects. Each **os_base_class** object represents a virtual base class from which the specified class inherits (that is, whose storage is allocated as part of the specified class). The order of list elements is not significant. err_mop_forward_definition is signaled if the specified class is known only through a forward definition.

os_class_type::get_base_classes()

os_list get_base_classes() const;

Returns a list, in declaration order, of pointers to **const os_base_class** objects. Each **os_base_class** object represents a class from which the given class is derived, together with the nature of the derivation (virtual or nonvirtual, and public, private, or protected). err_mop_forward_definition is signaled if the specified class is known only through a forward definition.

os_list get_base_classes();

Returns a list, in declaration order, of pointers to non-**const os_base_class** objects. Each **os_base_class** object represents a class from which the given class is derived, together with the nature of the derivation (virtual or nonvirtual, and public, private, or protected). err_mop_forward_definition is signaled if the specified class is known only through a forward definition.

os_class_type::get_class_kind()

os_unsigned_int32 get_class_kind() const;

Returns an enumerator indicating the kind of class represented by the specified instance of **os_class_type**. **os_class_type::Struct** indicates a struct; **os_class_type::Union** indicates a named union type; **os_class_type::Anonymous_union** indicates an anonymous union type; and **os_class_type::Class** indicates a class declared with the class-key **class**.

os_class_type::get_dispatch_table_pointer_offset()

os_int32 get_dispatch_table_pointer_offset() const;

Returns the offset at which the pointer to the dispatch table is stored. Signals err_mop if there is no dispatch table.

os_class_type::get_indirect_virtual_base_classes()

os_list get_indirect_virtual_base_classes() const;

Returns a list of pointers to **const os_base_class** objects. Each **os_ base_class** object represents a virtual base class from which the specified class inherits virtually and indirectly. The order of list elements is not significant. err_mop_forward_definition is signaled if the specified class is known only through a forward definition.

os_list get_indirect_virtual_base_classes();

Returns a list of pointers to non-**const os_base_class** objects. Each **os_base_class** object represents a virtual base class from which the specified class inherits virtually and indirectly. The order of list elements is not significant. err_mop_forward_definition is signaled if the specified class is known only through a forward definition.

os_class_type::get_members()

os_list get_members() const;

Returns a list, in declaration order, of pointers to **const os_member** objects. Each **os_member** object represents a member defined by the specified class. Note that currently only discriminant member functions are stored in the schema. err_mop_forward_definition is signaled if the specified class is known only through a forward definition.

os_list get_members();

Returns a list, in declaration order, of pointers to non-**const os_ member** objects. Each **os_member** object represents a member defined by the specified class. Note that currently only discriminant member functions are stored in the schema. err_mop_ forward_definition is signaled if the specified class is known only through a forward definition.

os_class_type::get_most_derived_class()

```
static const os_class_type &get_most_derived_class(
    const void *object,
    const void* &most_derived_object
) const;
```

If **object** points to the value of a data member for some other object, o, this function returns a reference to the *most derived* class of which o is an instance. A class, c1, is more derived than another class, c2, if c1 is derived from c2, or derived from a class derived from c2, and so on. **most_derived_object** is set to the beginning of the instance of the most derived class. There is one exception to this behavior, described below.

If **object** points to an instance of a class, o, but not to one of its data members (for example, because the memory occupied by the instance begins with a virtual table pointer rather than a data member value), the function returns a reference to the most derived class of which o is an instance. **most_derived_object** is set to the beginning of the instance of the most derived class. There is one exception to this behavior, described below.

If **object** does not point to the memory occupied by an instance of a class, **most_derived_object** is set to **0**, and err_mop is signaled. ObjectStore issues an error message like the following:

<err-0008-0010>Unable to get the most derived class in os_class_
type::get_most_derived_class() containing the address 0x%lx.

Here is an example:

```
class B {
public:
int ib ;
} ;

class D : public B {
public:
int id ;
} ;

class C {
public:
int ic ;
D cm ;
} ;

void baz () {
C* pC = new (db) C;
D *pD = &C->cm ;
int *pic = &pC->ic, *pid = &pC->cm.id, *pib = &pC->cm.ib ;
...
}
```

Invoking **get_most_derived_class()** on the pointers **pic**, **pid**, and **pib** has the results shown in the following table:

object	**most_derived_object**	**os_class_type**
pic	pC	C
pid	pD	D
pib	pD	D

The exception to the behavior described above can occur when a class-valued data member is collocated with a base class of the class that defines the data member. If a pointer to such a data member (which is also a pointer to such a base class) is passed to **get_most_derived_class()**, a reference to the value type of the data member is returned, and **most_derived_object** is set to the same value as **object**.

Consider, for example, the following class hierarchy:

```
class C0 {
public:
int i0;
};

class B0 {
public:
void f0();
};

class B1 : public B0 {
public:
virtual void f1();
C0 c0;
};

class C1 : public B1 {
public:
static os_typespec* get_os_typespec();
int i1;
};
```

Some compilers will optimize **B0** so that it has zero size in **B1** (and **C1**). This means the class-valued data member **c0** is collocated with a base class, **B0**, of the class, **C1**, that defines the data member.

Given

```
    C1 c1;
    C1 * pc1 = & c1;
    B0 * pb0 = (B0 *)pc1;
    C0 * pc0 = & pc1->c0;
```

the pointers **pb0** and **pc0** will have the same value, because of this optimization.

In this case **get_most_derived_class()** called on the **pb0** or **pc0** will return a reference to the **os_class_type** for **C0** (the value type of the data member **c0**) and **most_derived_object** is set to the same value as **object**.

If **B1** is instead defined as

```
class B1 : public B0 {
public:
    virtual void f1();
    int i;
    C0 c0;
};
```

and

```
    int * pi = & pc1->i;
```

pb0 and **pi** have the same value because of the optimization, but the base class, **B0**, is collocated with an **int**-valued data member rather than a class-valued data member. **get_most_derived_class()** called on **pb0** or **pi** returns a pointer to the **os_class_type** for the class **C1** and sets **most_derived_object** to the same value as **pc1**.

os_class_type::get_name()

```
const char *get_name() const;
```

Returns the name of the specified class.

os_class_type::get_pragmas()

```
os_List<os_pragma*> get_pragmas() const;
```

Returns the pragmas associated with the specified class.

os_class_type::get_size_as_base()

```
os_unsigned_int32 get_size_as_base() const;
```

Returns the size of the specified class when serving as a base class.

os_class_type::get_source_position()

```
void get_source_position(
    const char* &file,
    os_unsigned_int32 &line
) const;
```

Returns the source position associated with the specified class.

os_class_type::has_constructor()

```
os_boolean has_constructor() const;
```

Returns nonzero if the class defines a constructor.

os_class_type::has_destructor()

```
os_boolean has_destructor() const;
```

Returns nonzero if the class defines a destructor.

os_class_type::has_dispatch_table()

```
os_boolean has_dispatch_table() const;
```

Returns nonzero if the class has a dispatch table.

os_class_type::introduces_virtual_functions()

```
os_boolean introduces_virtual_functions() const;
```

Returns nonzero if and only if the class defines, rather than merely inherits, a virtual function.

You can set this attribute with **set_introduces_virtual_functions()**.

os_class_type::is_abstract()

```
os_boolean is_abstract() const;
```

Returns nonzero if and only if the specified class is abstract.

os_class_type::is_forward_definition()

```
os_boolean is_forward_definition() const;
```

Returns **1** if and only if the specified class is known only through a forward definition.

os_class_type::is_persistent()

```
os_boolean is_persistent() const;
```

Returns nonzero if and only if the specified class is marked as persistent.

os_class_type::is_template_class()

os_boolean is_template_class() const;

Returns **1** if and only if the specified class is a template class.

os_class_type::operator os_instantiated_class_type&()

operator const os_instantiated_class_type&() const;

Provides for safe casts from **os_class_type to const os_ instantiated_class type.** If the cast is not permissible, err_mop_ illegal_cast is signaled.

operator os_instantiated_class_type&();

Provides for non-**const** casts from **os_class_type to os_ instantiated_class type.** If the cast is not permissible, err_mop_ illegal_cast is signaled.

os_class_type::set_access_of_get_os_typespec_function()

```
void set_access_of_get_os_typespec_function(
    os_member::os_member_access
);
```

Pass **os_member::Private**, **os_member::Protected**, or **os_ member::Public** to specify the type of access allowed to the class’s **get_os_typespec()** function.

os_class_type::set_base_classes()

void set_base_classes(os_List<os_base_class*>&);

Sets the list, in declaration order, of **os_base_class**es of the specified class. Each **os_base_class** object represents a class from which the given class is derived, together with the nature of the derivation (virtual or nonvirtual, and public, private, or protected).

os_class_type::set_class_kind()

void set_class_kind(os_unsigned_int32);

Sets the class kind of the specified class. The argument should be an enumerator indicating the kind of class represented by the specified instance of **os_class_type**. **os_class_type::Struct** indicates a struct; **os_class_type::Union** indicates a named union type; **os_class_type::Anonymous_union** indicates an anonymous union type; and **os_class_type::Class** indicates a class declared with the class-key **class**.

os_class_type::set_declares_get_os_typespec_function()

void set_declares_get_os_typespec_function(os_boolean);

Specifies whether **this** declares a **get_os_typespec()** member function.

os_class_type::set_defines_virtual_functions()

void set_defines_virtual_functions(os_boolean);

Specifies whether the specified class defines any virtual functions.

os_class_type::set_dispatch_table_pointer_offset()

void set_dispatch_table_pointer_offset(os_int32);

Sets the offset at which the pointer to the dispatch table is stored.

os_class_type::set_has_constructor()

void set_has_constructor(os_boolean);

Specifies whether the class defines a constructor.

os_class_type::set_has_destructor()

void set_has_destructor(os_boolean);

Specifies whether the class defines a destructor.

os_class_type::set_indirect_virtual_base_classes()

void set_indirect_virtual_base_classes(os_List<os_base_class*>);

Sets the indirect virtual base classes of the specified class.

os_class_type::set_introduces_virtual_functions()

void set_introduces_virtual_functions(os_boolean);

Passing a nonzero value specifies that the class defines, rather than merely inherits, a virtual function.

os_class_type::set_is_abstract()

void set_is_abstract(os_boolean);

Specifies whether the specified class is abstract.

os_class_type::set_is_forward_definition()

void set_is_forward_definition(os_boolean);

Specifies whether the specified class is known only through a forward definition.

os_class_type::set_is_persistent()

void set_is_persistent(os_boolean);

Specifies whether the specified class is marked as persistent. In order to be installed into a database schema, a class must either be marked as persistent or be reachable from a persistent class. Making a class persistent with **set_is_persistent()** is similar to marking it with **OS_MARK_SCHEMA_TYPE()**.

os_class_type::set_members()

void set_members(os_List<os_member*>&);

Sets the members, in declaration order, of the specified class.

os_class_type::set_name()

void set_name(const char *);

Sets the name of the specified class. ObjectStore copies the character array pointed to by the argument.

os_class_type::set_pragmas()

void set_pragmas(os_List<os_pragma*>);

Sets the pragmas associated with the specified class.

os_class_type::set_source_position()

```
void set_source_position(
    const char* file,
    os_unsigned_int32 line
);
```

Sets the source position associated with the specified class.

os_class_type::Struct

This enumerator is a possible return value from **os_class_type::kind()**. It indicates a struct.

os_class_type::Union

This enumerator is a possible return value from **os_class_type::kind()**. It indicates a named union type.

os_comp_schema

This class is part of the ObjectStore metaobject protocol, which provides access to ObjectStore schemas. An instance of this class represents a compilation schema, stored in a compilation schema database. **os_comp_schema** is derived from **os_schema**.

Programs using this class must include **<ostore/ostore.hh>**, followed by **<ostore/coll.hh>** (if used), followed by **<ostore/mop.hh>**.

os_comp_schema::get()

static const os_comp_schema &get(const os_database&);

Returns the schema of the specified database.

os_database

Instances of the class **os_database** represent ObjectStore databases. Pointers to such instances can be used as arguments to persistent **new**.

Persistent data can be accessed only if the database in which it resides is open. Databases are created, destroyed, opened, and closed with database member functions. The functions for creating, opening, and closing databases can be called either inside or outside a transaction. The function for destroying databases should be called outside a transaction.

Each database retrieved by a given process has an associated *open count* for that process. The member function **os_database::open()** increments the open count by 1, and the member function **os_database::close()** decrements the open count by 1. When the open count for a process increases from **0** to **1**, the database becomes open for that process. A database becomes closed for a process when its open count becomes **0**.

When a process terminates, any databases left open are automatically closed.

Instances of the class **os_database** are sometimes used to hold *process local* or *per-process* state, representing a property of the database *for the current process*. For example, the function **os_database::is_writable()** returns a value indicating whether the specified database is writable for the current process. This is per-process information since the database might be unwritable by one process (because it opened the database for read only — see **os_database::open()** on page 88) while it is writable by another process. As a consequence, it is sometimes preferable to think of an instance of this class as representing an *association* between a database and a process (the process that retrieves the pointer to it). In fact, instances of **os_database** are actually transient objects.

Invoking **::operator delete()** on an **os_database** is illegal.

Some functions that perform administrative operations on databases (for example functions for changing database ownership or protection modes) are members of the class **os_dbutil** — see a description on page 106.

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. The type **os_unixtime_t** is defined as **unsigned long**.

All ObjectStore programs must include the header file **<ostore/ostore.hh>**.

os_database::allow_external_pointers()

void allow_external_pointers(os_boolean set_default_only = 0);

Must be called from within a transaction. Once invoked, cross-database pointers are allowed from all current and future segments of the specified database, unless **set_default_only** is specified as a nonzero value (true), in which case cross-database pointers will be allowed only from subsequently created segments.

A pointer from one database, db1, to another, db2, points to, at any given time, whichever database has a certain pathname — namely, db2’s pathname at the time the pointer was stored. If db2’s pathname changes (for example, as a result of performing **osmv** on db2), the pointer will no longer refer to data in db2. If some other database, db3, is given db2’s original pathname (for example, as a result of performing **osmv** on db3), the pointer will refer to data in db3.

The pathname is not stored as part of the cross-database pointer (which takes the form of a regular virtual memory address), but rather as part of an **os_database_reference** stored in a table associated with the pointer.

It is illegal to rename a database so that a pointer that used to refer to another database now refers to the database in which the pointer itself resides.

References normally store a relative pathname; that is, if the source and destination databases have a common ancestor directory, the common directory is not stored as part of the pathname — only the part of the target database’s pathname that is not shared with the source database’s pathname is stored. On UNIX systems, for example, the common part of the pathname is preceded by the appropriate number of “**../**”s to traverse the hierarchy up from the source directory to the common ancestor directory. For example, if the source and target databases are named "**/sys/appl/mydb**" and "**/sys/lib/lib1**" respectively, the reference stores the relative pathname, "**../lib/lib1**".

You can override use of relative pathnames with the functions **os_database::set_relative_directory()** and **os_database::get_relative_directory()** — see the entries for these functions below.

Dereferencing a cross-database pointer causes the destination database, if not already open, to be opened for read / write. Thus, dereferencing such a pointer can result in err_database_not_found.

os_database::change_database_reference()

```
void change_database_reference(
    os_database_reference *old_ref,
    os_database_reference *new_ref
);
```

Substitutes **new_ref** for **old_ref** in all tables that resolve pointers out of the specified database. Can be used to change the type of a cross-database pointer's associated **os_database_reference**. Does not affect pointers already resolved. This function signals err_trans if it is called within a transaction.

os_database::change_database_reference starts its own transaction (outside any user transaction), and iterates over every segment in the database to find and change the reference. Be aware that there is a small risk of an address space problem if there are **pvar**s that trigger data page use during the internal transaction that **os_database::change_database_reference** uses.

See also the functions **os_database::allow_external_pointers()** on page 72 and **os_database::get_database_references()** on page 81, and the class **os_database_reference** on page 101.

os_database::change_schema_key()

```
void change_schema_key(
    os_unsigned_int32 old_key_low,
    os_unsigned_int32 old_key_high,
    os_unsigned_int32 new_key_low,
    os_unsigned_int32 new_key_high
);
```

Sets the schema key of the specified database. Call this function from within an update transaction. The specified database must be opened for update, otherwise ObjectStore signals err_opened_read_only, and issues an error message like the following:

<err-0025-0155> Attempt to change the schema key of database db1, but it is opened for read only.

If the database has had its key frozen, err_schema_key is signaled, and ObjectStore issues an error message like the following:

err_schema_key

<err-0025-0152> The schema key of database db1 is frozen and may not be changed.

If the database already has a schema key at the time of the call, **old_key_low** must match the first component of the key and **old_key_high** must match the second component, or err_schema_key is signaled, and ObjectStore issues an error message like the following:

Error using schema keys
<err-0025-0158>Unable to change schema key of database db1.
The schema is already protected and the key provided did not match the old key in the schema. (err_schema_key)

If the database has no schema key, **old_key_low** and **old_key_high** are ignored.

new_key_low specifies the first component of the database's new schema key, and **new_key_high** specifies the second component. If both these arguments are **0**, calling this function causes the database to have no schema key.

os_database::close()

void close();

If the open count of the database for which the function is called is greater than **0**, decrements the open count by **1**. If the function is called from within a transaction, the open count is not decremented until the end of the current outermost transaction. If the open count becomes **0**, the database is closed. If the new open count, **c**, remains greater than **0**, the database is given the access type (opened for read or opened for read / write) it had as of the last time the open count was **c**.

os_database::create()

static os_database *create(

```
    const char *pathname,
    os_int32 mode = 0664,
    os_boolean if_exists_overwrite = 0,
    os_database *schema_database = 0
);
```

Returns a pointer to a newly created database, an instance of the class **os_database**, with the specified **pathname** and **mode** (the values of **mode** are as described in *ObjectStore Management*, oschmod: Changing Database Permissions). The new database is also opened for read / write, and its open count is incremented.

If a database with the specified pathname already exists, an err_database_exists exception is signaled, unless **if_exists_overwrite** is nonzero (true). If **if_exists_overwrite** is nonzero, any existing database with the same pathname is deleted, a new database is created with the specified mode, and the new database is opened for read / write.

If **schema_database** is **0**, schema information is stored in the new database. The effect is the same as the result of calling **create()** without the **schema_database** argument:

```
    os_database::create(pathname, mode, if_exists_overwrite);
```

If **schema_database** is nonzero, the database it specifies is used as the *schema database* for the newly created database. This means that ObjectStore installs in the schema database all schema information for the data stored in the new database; the new database itself will have no schema information in it.

The specified schema database must be open at the time of the call to **create()**; if it is not, err_schema_database is signaled. If the schema database was opened for read only, ObjectStore attempts to reopen it for read / write. If this fails because of protections on the database, it remains open for read only. This prevents any update to the new database that requires schema installation.

Note that the new database's schema database can also contain regular user data (that is, data other than schema information). The schema database must store its own schema locally. If the schema for the user data in **schema_database** is stored remotely, err_schema_database is signaled.

For file databases, **pathname** is an operating system pathname. ObjectStore takes into account local NFS mount points when interpreting the pathname, so pathnames can refer to databases on foreign hosts. To refer to a database on a foreign host for which there is no local mount point, use a Server host prefix, the name of the foreign host followed by a colon (:), as in **oak:/foo/bar**.

For databases in ObjectStore directories, **pathname** consists of a rooted pathname preceded by a rawfs host prefix of the form **host-name::** (for example **oak::/foo/bar**). The rawfs host prefix can be followed by a Server host prefix of the form **host-name:** (as in **oak::beech:/foo/bar**).

The Server host name specifies the file system on which you want the new database stored. If no Server host name is supplied, the value of the ObjectStore environment variable **OS_SERVER_HOST** is used.

os_database::create_root()

os_database_root *create_root(char *name);

Creates a root in the specified database to associate a copy of the specified name with an as-yet-unspecified entry-point object. (Since the name is copied to persistent memory by this function, the supplied **char*** can point to transient memory.) If the specified name is already associated with an entry-point object in the same database, an err_root_exists exception is signaled. If the specified database is the transient database, an err_database_not_open exception is signaled.

os_database::create_segment()

os_segment *create_segment();

Creates a segment in the specified database, and returns a pointer to an instance of the class **os_segment**. Call this function from within a transaction. The return value points to a transient object representing the new segment. Note that, since it points to transient memory, the return value of this function cannot be stored persistently. Performing this function on the transient database returns a pointer to the transient segment.

os_database::decache()

void decache();

This function deletes all internal storage associated with the given database. Its purpose is to allow applications to open large numbers of databases without continuing to use more virtual memory and commseg space for each database opened.

Calling **decache()** on an **os_database** makes the **os_database** object, and any **os_segment** objects for that database, unusable. You should not refer to the **os_database** or **os_segment** objects again; doing so could lead to unpredictable results. If you want to open or look up the database again, use **os_database::lookup()** or the overloading of **os_database::open()** that takes a pathname argument.

There are some restrictions on when this function can be used. The call to **os_database::decache()** must be made outside a transaction. The application cannot be in **retain_persistent_ addresses** mode. The database being decached cannot be the transient database, be currently opened by the application, or have open the database that it wants to decache. ObjectStore will raise an err_misc exception if any of these restrictions is violated.

Calling **decache()** on an **os_database** invalidates any **os_ reference_transient** objects that refer to objects within the database. Attempting to resolve these invalidated references will have unpredictable results.

Applications should only call this routine if they probably will not refer to the database soon. There is some overhead associated with using a database for the first time that will be incurred again if the database is opened after it is decached.

os_database::destroy()

void destroy();

Deletes the database for which the function is called. To make your program portable, only call this function from outside any transaction. If the database is open at the time of the call, **destroy()** closes the database before deleting it.

When a process destroys a database, this can affect other processes that have the database opened. Such a process might subsequently be unable to access some of the database’s data — even if earlier in the same transaction it successfully accessed the database.

Data already cached in the process's client cache will continue to be accessible, but attempts to access other data will cause ObjectStore to signal err_database_not_found. Attempts to open the database will also provoke err_database_not_found. Note that performing **os_database::lookup()** on the destroyed database's pathname might succeed, since the instance of **os_database** representing the destroyed database might be in the process's client cache.

If you call this function from within a transaction, beware of the following:

- The effects of calling this function cannot be undone by aborting the transaction.
- On some but not all platforms, calling **destroy()** from within a transaction causes ObjectStore to signal the exception err_database_lock_conflict if another process is accessing the database.
- If you attempt to access data in a destroyed database in the same transaction in which it was destroyed, err_database_not_found is signaled.
- In most cases the database is actually removed from the Server at the end of the transaction, but in some cases it is removed earlier. For example, suppose that, before the transaction ends, you create a new database with the same pathname as the database on which you called **destroy()**. In this case, the old database is removed before the new database is created.

If you attempt to operate on a destroyed instance of **os_database**, err_database_is_deleted is signaled.

os_database::find_root()

```
os_database_root *find_root(char *name);
```

Returns a pointer to the root in the specified database with the specified name. Returns **0** if not found.

os_database::freeze_schema_key()

```
void freeze_schema_key(
    os_unsigned_int32 key_low,
    os_unsigned_int32 key_high
);
```

Freezes the specified database's schema key, preventing any change to the key, even by applications with a matching key.

Call this function from within an update transaction. The specified database must be opened for update, otherwise ObjectStore signals err_opened_read_only, and issues an error message like the following:

<err-0025-0156> Attempt to freeze the schema key of database db1, but it is opened for read only.

If the database is schema protected and has not been accessed since the last time its open count was incremented from **0** to **1**, the application's schema key must match the database's schema key. If it does not, err_schema_key is signaled, and ObjectStore issues an error message like the following:

<err-0025-0151>The schema is protected and the key, if provided, did not match the one in the schema of database dba.

key_low and **key_high** must match the database's schema key, or else err_schema_key is signaled, and ObjectStore issues an error message like the following:

<err-0025-0159>Unable to freeze the schema key of database db1. The schema is protected and the key provided did not match the key in the schema.

If the database's schema key is already frozen, and you specify the correct key, the call has no effect.

os_database::get_all_databases()

```
static void get_all_databases(
    os_int32 max_to_return,
    os_database_p *dbs,
    os_int32& n_ret
);
```

Provides access to all the databases retrieved by the current process. The **os_database_p*dbs** is an array of pointers to databases. This array must be allocated by the user. The function **os_database::get_n_databases()** can be used to determine how large an array to allocate. **max_to_return** is specified by the user, and is the maximum number of elements the array is to have. **n_ret** refers to the actual number of elements in the array.

os_database::get_all_roots()

```
void get_all_roots(
    os_int32 max_to_return,
    os_database_root_p *roots,
    os_int32& n_ret
);
```

Provides access to all the roots in the specified database (see **os_ database_root** on page 103). The **os_database_root_p*** is an array of pointers to roots. This array must be allocated by the user. The function **os_database::get_n_roots()** can be used to determine how large an array to allocate. **max_to_return** is specified by the user, and is the maximum number of elements the array is to have. **n_ret** refers to the actual number of elements in the array.

os_database::get_all_segments()

```
void get_all_segments(
    os_int32 max_to_return,
    os_segment_p *segs,
    os_int32& n_ret
);
```

iProvides access to all the segments in the specified database. The **os_segment_p*** is an array of pointers to segments. This array must be allocated by the user. The function **os_database::get_n_ segments()** can be used to determine how large an array to allocate. **max_to_return** is specified by the user, and is the maximum number of elements the array is to have. **n_ret** refers to the actual number of segment pointers returned.

os_database::get_all_segments_and_permissions()

This member of **os_database** is declared as follows:

```
void get_all_segments_and_permissions(
    os_int32 max_to_return,
    os_segment_p* segs,
    os_segment_access_p* controls,
    os_int32 &n_returned
);
```

Provides access to all the segments in the specified database, together with each segment's associated **os_segment_access**. The *n*th element of **controls** points to the **os_segment_access** associated with the segment pointed to by the *n*th element of **segs**.

The arrays **controls** and **segs** must be allocated by the user. **max_ to_return** is specified by the user.

os_database::get_application_info()

void *get_application_info() const;

Returns a pointer to the object pointed to by the pointer last passed, during the current process, to **os_database::set_ application_info()** for the specified database. If **set_application_ info()** has not been called for the specified database during the current process, **0** is returned.

os_database::get_check_illegal_pointers()

os_boolean get_check_illegal_pointers() const;

Returns nonzero if the database is in **check_illegal_pointers** mode; returns **0** otherwise.

os_database::get_database_references()

```
void get_database_references(
    os_int32 &n_refs,
    os_database_reference_p *&array
) const;
```

Allocates an array of database references on the heap, one for each database referenced by the specified database. When the function returns, **n_refs** refers to the number of elements in the array. Note that it is the user's responsibility to deallocate the array when it is no longer needed.

os_database::get_default_check_illegal_pointers()

os_boolean get_default_check_illegal_pointers() const;

Returns nonzero if the specified database is in **default_check_ illegal_pointers** mode; returns **0** otherwise. See **os_database::set_ default_check_illegal_pointers()** on page 94.

os_database::get_default_lock_whole_segment()

objectstore_lock_option get_default_lock_whole_segment() const;

Indicates the locking behavior for segments newly created in the specified database. **objectstore_lock_option** is an enumeration type whose enumerators are **objectstore::lock_as_used**, **objectstore::lock_segment_read**, and **objectstore::lock_segment_**

write. See **os_database::set_default_lock_whole_segment()** on page 94.

os_database::get_default_null_illegal_pointers()

os_boolean get_default_null_illegal_pointers() const;

Returns nonzero if the specified database is in **default_null_illegal_pointers** mode; returns **0** otherwise. See **os_database::set_default_null_illegal_pointers()** on page 94.

os_database::get_default_segment()

os_segment *get_default_segment() const;

Returns a pointer to the *default segment* of the specified database. The default segment is the segment in which persistent memory is allocated by default, when the function **new** is called with only an **os_database*** argument. Initially the default segment is the *initial segment*, the one segment (besides the schema segment) with which the database was created. A process can change this at any time (see **os_database::set_default_segment()** on page 94), but the change remains in effect only for the duration of the process, and is invisible to other processes. Simple ObjectStore applications need not create any segments; all the database's persistent data can be stored in the initial segment, if desired. But if more sophisticated clustering is required, the application can create new segments in the database, and it might be convenient to make one of these the default.

os_database::get_default_segment_size()

os_int32 get_default_segment_size() const;

The initial size in bytes of segments in the specified database. See **os_database::set_default_segment_size()** on page 95.

os_database::get_dirman_host_name()

char *get_dirman_host_name() const;

If the specified database is a rawfs database, the function allocates on the heap and returns the name of the host machine for the rawfs. If the database is a file database or the transient database, **0** is returned. Note that it is the user's responsibility to deallocate the character array when it is no longer needed.

os_database::get_fetch_policy()

void get_fetch_policy(os_fetch_policy &policy, os_int32 &bytes);

Sets **policy** and **bytes** to references to an **os_fetch_policy** and integer that indicate the database's current fetch policy. See **os_database::set_fetch_policy()** on page 95.

os_database::get_file_host_name()

char *get_file_host_name() const;

If the specified database is a file database, the function allocates on the heap and returns the name of the host machine for the database. If the database is a rawfs database or the transient database, **0** is returned. Note that it is the user's responsibility to deallocate the character array when it is no longer needed.

os_database::get_host_name()

char *get_host_name() const;

Returns the name of the host machine on which the specified database resides. The returned **char*** points to an array allocated on the heap by this function. Note that it is the user's responsibility to deallocate the character array when it is no longer needed.

os_database::get_id()

os_database_id *get_id() const;

Returns the **os_database_id** of the specified database.

os_database::get_incremental_schema_installation()

os_boolean get_incremental_schema_installation();

Returns nonzero (true) if the schema installation mode of the specified database is set to incremental mode. Returns **0** (false) if the mode is set to batch mode (the default). See **os_database::set_incremental_schema_installation()** on page 96.

os_database::get_lock_whole_segment()

objectstore_lock_option get_lock_whole_segment() const;

Indicates the locking behavior currently in effect for the specified database. **objectstore_lock_option** is an enumeration type whose enumerators are **objectstore::lock_as_used**, **objectstore::lock_segment_read**, and **objectstore::lock_segment_write**. See **os_segment::set_lock_whole_segment()** on page 303.

os_database::get_n_databases()

static os_int32 get_n_databases();

Returns the number of databases retrieved by the current process.

os_database::get_n_roots()

os_int32 get_n_roots() const;

Returns the number of roots in the specified database (see **os_database_root** on page 103).

os_database::get_n_segments()

os_int32 get_n_segments() const;

Returns the number of segments in the specified database, including the schema segment.

os_database::get_opt_cache_lock_mode()

os_boolean get_opt_cache_lock_mode() const;

Returns nonzero if **opt_cache_lock_mode** is on for the current process and the specified database; returns **0** otherwise. See **os_database::set_opt_cache_lock_mode()** on page 98.

os_database::get_pathname()

char *get_pathname() const;

Returns the pathname of the specified database. For databases in ObjectStore directories, the pathname consists of a rawfs host prefix ("**rawfs-host-name::**") followed by a rooted pathname (for example, "**oak::/parts/db1**"). For file databases, the pathname is identical to the one passed to **os_database::open()**, **os_database::create()**, or **os_database::lookup()** when **this** was first retrieved by the current process. The returned **char*** points to an array allocated on the heap by this function. Note that it is the user's responsibility to deallocate the array when it is no longer needed.

os_database::get_prms_are_in_standard_format()

os_boolean get_prms_are_in_standard_format() const;

os_database::get_readlock_timeout()

os_int32 get_readlock_timeout() const;

Returns the time in milliseconds for which the current process will wait to acquire a read lock on pages in the specified database. A value of **–1**, the default, indicates that the process will wait forever if necessary.

os_database::get_relative_directory()

char *get_relative_directory() const;

Returns a string indicating the current method used for storing database references. If **0** (null) is returned, the default method is used. If an empty string is returned, rooted pathnames are used. Otherwise, the string returned indicates the name of a directory to be treated as a common ancestor to the source and destination databases. On UNIX systems, for example, the common part of the pathnames is preceded by the appropriate number of "**../**"s to traverse the hierarchy up from the source directory to this common ancestor directory. If the indicated directory is not actually a common ancestor, a rooted pathname is used. The returned **char*** points to an array allocated on the heap by this function. Note that it is the user's responsibility to deallocate the array when it is no longer needed. See **os_database::allow_external_pointers()** on page 72 and **os_database::set_relative_directory()** on page 98.

os_database::get_required_DLL_identifiers()

Component Schema

```
const char* const* get_required_DLL_identifiers(
    os_unsigned_int32& count
);
```

Returns an array of pointers to DLL identifiers and the number of elements in the array. The order of elements in the array is not significant. The array and the elements must not be modified or deallocated.

This function can only be called within a transaction with the database open. The returned array and strings might reside in the database so they are only valid for one transaction.

os_database::get_schema_database()

os_database *get_schema_database() const;

Returns a pointer to the schema database for **this**. If **this** is not a database whose schema is stored remotely, **0** is returned. This function must be invoked within a transaction.

os_database::get_sector_size()

os_unsigned_int32 get_sector_size();

Returns the size of a sector, in bytes.

os_database::get_segment()

os_segment *get_segment(os_unsigned_int32 segment_number);

Returns a pointer to the segment in the specified database with the specified segment number. See **os_segment::get_number()** on page 299.

os_database::get_transient_database()

static os_database *const get_transient_database();

Returns a pointer to the transient database, which can be used to request transient memory allocation, for example as an argument to **new()**.

os_database::get_writelock_timeout()

os_int32 get_writelock_timeout() const;

Returns the time in milliseconds for which the current process will wait to acquire a write lock on pages in the specified database. A value of **–1**, the default, indicates that the process will wait forever if necessary.

os_database::insert_required_DLL_identifier()

Component schema

```
void insert_required_DLL_identifier(
    const char* DLL_identifier
);
```

Copies the **DLL_identifier** string and adds it to the database's set of required DLLs. If the **DLL_identifier** is already in the database's set of required DLLs, this function does nothing. Call this function only in an update transaction with the database open for write.

See also **os_database::insert_required_DLL_identifiers()**

os_database::insert_required_DLL_identifiers()

```
void insert_required_DLL_identifiers
    (const char* const* DLL_identifiers,
    os_unsigned_int32 count
);
```

Copies **DLL_identifiers** to a database's set of required DLLs.

For each identifier, checks that the identifier is not already in the set. If already present, the identifier is not copied. If the identifier isn't present in the set of required DLLs, it is copied and added to the set. This function can only be called in an update transaction with the database open for write.

os_database::insert_required_DLL_identifiers is equivalent to repeated calls to **os_database::insert_required_DLL_identifier** but is more efficient.

os_database::is_open()

```
os_boolean is_open() const;
```

Returns a nonzero **os_boolean** (true) if the database for which the function is called is open, and **0** (false) otherwise.

os_database::is_open_mvcc()

```
os_boolean is_open_mvcc() const;
```

Returns nonzero if this is opened for MVCC, and **0** otherwise. See **os_database::open_mvcc()** on page 89.

os_database::is_open_read_only()

```
os_boolean is_open_read_only() const;
```

Returns a nonzero **os_boolean** (true) if the database for which the function is called is open for read only, and **0** (false) otherwise.

os_database::is_writable()

```
os_boolean is_writable() const;
```

Returns a nonzero **os_boolean** (true) if the database for which the function is called is writable by the current process. The function returns **0** (false) if the database is not writable, for example because the current process opened it **read_only**, or because, due to access control, the process does not have write permission. The return value is not affected by whether the current transaction, if any, is a **read_only** transaction. If performed on a database that is not open, a run-time error is signaled.

os_database::lookup()

```
static os_database *lookup(
    const char *pathname
    os_int32 create_mode = 0
);
```

Returns a pointer to the database with the specified **pathname** (but does not open it). If not found, an err_database_not_found exception is signaled. **create_mode** is a boolean; if its value is nonzero and no database named **pathname** exists, it creates the database.

For information on database pathnames, see **os_database::create()** on page 74.

os_database::of()

```
static os_database *of(void *location);
```

Returns a pointer to the database in which the specified object resides. If the object is transiently allocated, a pointer to the transient database is returned. In almost all cases you should use **os_segment::of()** instead of **os_database::of()**. In particular, you should never use the result of **os_database::of()** to allocate a new persistent object. Doing so will defeat segment clustering. When in doubt, use **os_segment::of()**.

os_database::open()

```
void open(os_boolean read_only = 0);
```

Increases the open count of the specified database by 1, and establishes the access type specified by **read_only** — nonzero (true) for read only and **0** for read / write.

```
static os_database *open(
    const char *pathname,
    os_boolean read_only = 0,
    os_int32 create_mode = 0
);
```

Increments the open count of the database with the specified **pathname**, establishes the access type specified by **read_only** —

nonzero (true) for read only and **0** (false) for read / write — and returns a pointer to that database. If not found, an err_database_not_found exception is signaled, unless **create_mode** is nonzero.

If **create_mode** is nonzero and no database named **pathname** exists, the effect is the same as calling **os_database::create()** with the same **pathname**, **create_mode**, and **schema_database** arguments. The values of **create_mode** — which must be octal numbers, beginning with **0** — are described in *ObjectStore Management*.

For information on database pathnames, see **os_database::create()** on page 74.

```
static os_database *open(
    const char *pathname,
    os_boolean read_only,
    os_int32 create_mode,
    os_database *schema_database
);
```

The first three arguments are as described for the previous overloading of **open()**. If no database named **pathname** is found, and **schema_database** is nonzero, **schema_database** is used as the *schema database* for the newly created database. This means that ObjectStore installs in the schema database all schema information for the data stored in the new database; the new database itself will have no schema information in it.

The specified schema database must be open at the time of the call to **open()**; if it is not, err_schema_database is signaled. If the schema database was opened read only, ObjectStore attempts to reopen it for read / write. If this fails because of protections on the database, it remains open for read only. This prevents any update to the new database that requires schema installation.

Note that the new database's schema database can also contain regular user data (that is, data other than schema information). The schema database must store its own schema locally. If the schema for the user data in **schema_database** is stored remotely, err_schema_database is signaled.

os_database::open_mvcc()

```
void open_mvcc();
```

Opens a database for multiversion concurrency control (MVCC). Once you open a database for MVCC, multiversion concurrency control is used for access to it until you close it. If the database is already opened, but not for MVCC, err_mvcc_nested is signaled. If you try to perform write access on a database opened for MVCC, err_opened_read_only is signaled.

If an application has a database opened for MVCC, it never has to wait for locks to be released in order to read the database. Reading a database opened for MVCC also never causes other applications to have to wait to update the database. In addition, an application never causes a deadlock by accessing a database it has opened for MVCC.

In each transaction in which an application accesses a database opened for MVCC, the application sees what it would see if viewing a snapshot of the database taken *sometime* during the transaction. This snapshot has the following characteristics:

- It is internally consistent.
- It might not contain changes committed during the transaction by other processes.
- It does contain all changes committed before the transaction started.

Because of the second characteristic, the snapshot might not be consistent with other databases accessed in the same transaction (although it will always be internally consistent). Even two databases both of which are opened for MVCC might not be consistent with each other, because updates might be performed on one of the databases between the times of their snapshots.

Even though the snapshot might be out of date by the time some of the access is performed, multiversion concurrency control retains serializability, if each transaction that accesses an MVCC database accesses only that one database. Such a transaction sees a database state that would have resulted from some serial execution of all transactions, and all the transactions produce the same effects as would have been produced by the serial execution.

static os_database *open_mvcc(const char *pathname);

Opens the database with the specified pathname for multiversion concurrency control. See the first overloading of **open_mvcc()**, above.

os_database::remove_required_DLL_identifier()

Component schema

```
void remove_required_DLL_identifier(
    const char* DLL_identifier
);
```

Removes the **DLL_identifier** from the database's set of required DLLs. If **DLL_identifier** is not in the set, this function does nothing. This function can only be called in an update transaction with the database open for write.

os_database::set_access_hooks()

```
typedef void (*os_access_hook) (
    void *object,
    enum os_access_reason reason,
    void *user_data,
    void *start_range,
    void *end_range
);

void set_access_hooks
    os_char_p class_name,
    os_access_hook inbound_hook,
    os_access_hook outbound_hook,
    os_void_p user_data,
    os_access_hook *old_inbound_hook = 0,
    os_access_hook *old_outbound_hook = 0,
    os_void_p *old_user_data = 0
);
```

Registers an inbound and outbound hook function for the specified database and class.

The **class_name** argument is the name of the class with which these hook functions should be associated.

inbound_hook is the user-supplied hook function that is called each time an instance of the specified class in the specified database becomes accessible (that is, the first time in each transaction the object is accessed, as well as each time the object is transferred into the client cache).

outbound_hook is the user-supplied hook function that is called each time an instance of the specified class in the specified database undergoes outbound relocation. This typically occurs each time the transaction ends and the object is in the client cache, and each time the object is transferred out of the client cache.

In some cases, however, an object might be transferred out of the client cache without undergoing outbound relocation (see for example **OS_EVICT_IN_ABORT** in *ObjectStore Management*), and so the **outbound_hook** is not called. This means that there is not necessarily a call to the **outbound_hook** for each call to the **inbound_hook**. So you must structure your application to allow for consecutive calls to the inbound hook without intervening calls to the outbound hook.

The **inbound_hook** and **outbound_hook** arguments can be null pointers in order to specify that there should be no hook. To disable both hooks, pass a null pointer for both these arguments.

user_data is a pointer to user data to be passed to the hook functions.

The previous value for the **inbound_hook** is returned in the location pointed to by **old_inbound_hook**, if **old_inbound_hook** is a nonnull pointer. The previous value will be **0** if no **inbound_hook** was enabled. Similarly, the previous values of **outbound_hook** and **user_data** are returned in the locations pointed to by **old_outbound_hook** and **old_user_data** if these are nonnull pointers.

The hook functions are called with the following arguments:

object points to the beginning of the object for which the access hook is registered.

For an inbound hook, **reason** is always set to **os_reason_accessed**. For an outbound hook, **reason** is **os_reason_committed** if the object is on a page that is being relocated out at the end of a transaction, **os_reason_aborted** if the page is being relocated out during a transaction abort, and **os_reason_returned** if the page is being relocated out at some other time, presumably to make space in the cache for other data.

user_data is set to the value of **user_data** that was passed to **os_database::set_access_hooks()**.

start_range is the first address that is being made accessible during an inbound hook, or made inaccessible during an outbound hook. **end_range** is the first address beyond the range being made accessible or inaccessible. That is, all addresses greater than or equal to **start_range** and less than **end_range** are being made accessible or inaccessible.

Only a portion of the object might lie within **start_range** and **end_ range**. The hook should only write within this address range. You should make all modifications by calling **objectstore::hidden_ write()**. Note that the active region of the object might be writable in the context of the hook, but under no circumstances should the application modify any persistent addresses from within the hook by any other means other than by using the **objectstore::hidden_ write()** function.

Read access is allowed to any part of the object, not just the part within the address range. You cannot safely dereference pointers outside the object, but you can pass them to **objectstore::get_ pointer_numbers()**.

Classes within unions will not have access hooks called.

os_database::set_application_info()

void set_application_info(void *info);

Associates the specified object with the specified database for the current process. The argument **info** must point to a transient object. See **os_database::get_application_info()** on page 81.

os_database::set_check_illegal_pointers()

void set_check_illegal_pointers(os_boolean);

Provides a convenient way to set **check_illegal_pointers** mode for every segment in the specified database. If the argument is **1** (that is, true), it loops over every segment, including internal segments, enabling **check_illegal_pointers**. It also enables **default_check_ illegal_pointers** mode for the specified database. If the argument is **0** (that is, false), it disables **check_illegal_pointers** mode for every segment, and disables **default_check_illegal_pointers** mode for the specified database. See **os_segment::set_check_illegal_pointers()** on page 301 and **os_database::set_default_check_illegal_ pointers()** on page 94.

os_database::set_default_check_illegal_pointers()

void set_default_check_illegal_pointers(os_boolean);

If the argument is **1** (that is, true), this enables **default_check_ illegal_pointers** mode for the specified database. In this mode, new segments are created in **check_illegal_pointers** mode. If the argument is **0** (that is, false), this disables **default_check_illegal_ pointers** mode, and new segments are created with **check_illegal_ pointers** disabled. By default, **default_check_illegal_pointers** mode is disabled. See also **objectstore::set_check_illegal_pointers()** on page 35.

os_database::set_default_lock_whole_segment()

void set_default_lock_whole_segment(objectstore_lock_option);

Specifies the locking behavior for segments newly created in the specified database. See **os_segment::set_lock_whole_segment()** on page 303.

os_database::set_default_null_illegal_pointers()

void set_default_null_illegal_pointers(os_boolean);

If the argument is **1** (that is, true), this enables **default_null_illegal_ pointers** mode for the specified database. In this mode, new segments are created in **null_illegal_pointers** mode. If the argument is **0** (that is, false), this disables **default_null_illegal_ pointers** mode, and new segments are created with **null_illegal_ pointers** disabled. By default, **default_null_illegal_pointers** mode is disabled. See also **objectstore::set_null_illegal_pointers()** on page 38.

os_database::set_default_segment()

void set_default_segment(os_segment*);

Sets the *default segment* for the specified database. The default segment is the segment in which persistent memory is allocated by default, when the function **new** is called with only an **os_ database*** argument. Initially the default segment is the *initial segment*, the one segment (besides the schema segment) with which the database was created. Changing the default segment remains in effect only for the duration of the process, and is invisible to other processes. Simple ObjectStore applications need not create any segments; all the database's persistent data can be stored in the initial segment, if desired. But if more sophisticated clustering is required, the application can create new segments in the database, and it might be convenient to make one of these the default.

os_database::set_default_segment_size()

void set_default_segment_size(os_int32);

The specified **os_int32** is used to determine the initial size of segments in the specified database. This size is either the specified value (in bytes) or an implementation-dependent minimum size, whichever is larger.

os_database::set_fetch_policy()

enum os_fetch_policy { os_fetch_segment, os_fetch_page, os_ fetch_stream };

void set_fetch_policy(os_fetch_policy policy, os_int32 bytes);

Specifies the fetch policy for all the segments in the specified database. The policy argument should be one of the following enumerators: **os_fetch_segment**, **os_fetch_page**, or **os_fetch_ stream**.

The default fetch policy is **os_fetch_page**.

If an operation manipulates a substantial portion of a small segment, use the **os_fetch_segment** policy when performing the operation on the segment. Under this policy, ObjectStore attempts to fetch the entire segment containing the desired page in a single client / server interaction, if the segment will fit in the client cache without evicting any other data. If there is not enough space in the cache to hold the entire segment, the behavior is the same as for **os_fetch_page** with a fetch quantum specified by **bytes**.

If an operation uses a segment larger than the client cache, or does not refer to a significant portion of the segment, use the **os_fetch_ page** policy when performing the operation on the segment. This policy causes ObjectStore to fetch a specified number of bytes at a time, rounded up to the nearest positive number of pages, beginning with the page required to resolve a given object reference. **bytes** specifies the *fetch quantum*. (Note that if you specify **0** bytes, this will be rounded up, and the unit of transfer will be a single page.)

The default value for the fetch quantum is 4096 bytes (1 page). Appropriate values might range from 4 kilobytes to 256 kilobytes or higher, depending on the size and locality of the application data structures.

For special operations that scan sequentially through very large data structures, **os_fetch_stream** might considerably improve performance. As with **os_fetch_page**, this fetch policy lets you specify the amount of data to fetch in each client/server interaction for a particular segment. But, in addition, it specifies that a double buffering policy should be used to stream data from the segment.

This means that after the first two transfers from the segment, each transfer from the segment replaces the data cached by the *second-to-last* transfer from that segment. This way, the last two chunks of data retrieved from the segment will generally be in the client cache at the same time. And, after the first two transfers, transfers from the segment generally will not result in eviction of data from other segments. This policy also greatly reduces the internal overhead of finding pages to evict.

When you perform allocation that extends a segment whose fetch policy is **os_fetch_stream**, the double buffering described above begins when allocation reaches an offset in the segment that is aligned with the fetch quantum (that is, when the offset **mod** the fetch quantum is 0).

For all policies, if the fetch quantum exceeds the amount of available cache space (cache size minus wired pages), transfers are performed a page at a time. In general, the fetch quantum should be less than half the size of the client cache.

Note that a fetch policy established with **set_fetch_policy()** (for either a segment or a database) remains in effect only until the end of the process making the function call. Moreover, **set_fetch_policy()** only affects transfers made by this process. Other concurrent processes can use a different fetch policy for the same segment or database.

os_database::set_incremental_schema_installation()

void set_incremental_schema_installation(os_boolean);

If a nonzero **os_boolean** (true) is supplied as argument, the schema installation mode of the specified database is set to incremental mode. With incremental schema installation, a class is added to a database's schema only when an instance of that class is first allocated in the database. If **0** (false) is supplied as argument, the mode is set to batch mode (the default). With batch mode, whenever an application creates or opens the database, every class in the application's schema is added to the database's schema (if not already present in the database schema).

os_database::set_lock_whole_segment()

void set_lock_whole_segment(objectstore_lock_option);

Provides a convenient way to set **lock_whole_segment** mode for every segment in the specified database. It loops over every segment, including internal segments, setting **lock_whole_segment**, and setting **default_lock_whole_segment** in the database. See **os_segment::set_lock_whole_segment()** on page 303 and **os_database::set_default_lock_whole_segment()** on page 94.

os_database::set_new_id()

os_int32 set_new_id();

Changes the ID of the specified database to be a new unique ID.

os_database::set_null_illegal_pointers()

void set_null_illegal_pointers(os_boolean);

Provides a convenient way to set **null_illegal_pointers** mode for every segment in the specified database.

If the argument is **1** (that is, true), it loops over every segment, including internal segments, enabling **null_illegal_pointers**. It also enables **default_null_illegal_pointers** mode for the specified database.

If the argument is **0** (that is, false), it disables **null_illegal_pointers** mode for every segment, and disables **default_null_illegal_pointers** mode for the specified database.

See **os_segment::set_null_illegal_pointers()** on page 304 and **os_database::set_default_null_illegal_pointers()** on page 94.

os_database::set_opt_cache_lock_mode()

void set_opt_cache_lock_mode(os_boolean);

A nonzero argument turns on **opt_cache_lock_mode** for the current process and the specified database; a **0** argument turns the mode off.

Turning on this mode will improve performance for applications that perform writes to the specified database and expect little or no contention from other processes for access to the database. When this mode is on, the amount of client / server communication required to upgrade locks is reduced. Once a page from the database is cached on the client, the client can subsequently upgrade the page’s lock from read to read / write when needed without communicating with the Server. However, the amount of client / server communication required for concurrent processes to obtain locks on pages in the database might increase.

Note that this function sets the mode for the current process only, and does not affect the mode for other processes.

os_database::set_readlock_timeout()

void set_readlock_timeout(os_int32);

Sets the time in milliseconds for which the current process will wait to acquire a read lock on pages in the specified database. A value of **–1**, the default, indicates that the process should wait forever if necessary. After an attempt to acquire a read lock, if the specified time elapses without the lock’s becoming available, an os_lock_timeout_exception exception is signaled. If the attempt causes a deadlock, the transaction is aborted regardless of the value of the specified timeout.

os_database::set_relative_directory()

void set_relative_directory(char *dir_name);

Sets the method to be used for storing database references. Remains in effect only for the duration of the current process. If **0** (null) is supplied, the default method is used. If an empty string is supplied, rooted pathnames are used. Otherwise, the string supplied indicates the name of a directory to be treated as a common ancestor to the source and destination databases. The common part of the pathnames is preceded by the appropriate number of “**../**”s to traverse the hierarchy up from the source directory to this common ancestor directory. If the indicated directory is not actually a common ancestor, a rooted pathname is used. The contents of the specified character array are copied by this function. After this function returns, the array’s contents have no bearing on the relative directory. See **os_database::allow_ external_pointers()** on page 72 and **os_database::get_relative_ directory()** on page 85.

os_database::set_schema_database()

void set_schema_database(os_database& schema_database);

If you move a schema database, you must use **os_database::set_ schema_database()** or the ObjectStore utility **ossetrsp** to inform ObjectStore of the schema database’s new pathname. Calling this function establishes **schema_database** as the schema database for **this**. You must invoke this function outside any transaction.

If you copy the schema database with an operating system command or an ObjectStore utility, you can also use **os_ database::set_schema_database()** to establish the copy as the schema database for **this**. If **schema_database** is not the result of copying or moving a database that has served as schema database for **this**, err_schema_database is signaled.

os_database::set_writelock_timeout()

void set_writelock_timeout(os_int32);

Sets the time in milliseconds for which the current process will wait to acquire a write lock on pages in the specified database. A value of **–1**, the default, indicates that the process should wait forever if necessary. After an attempt to acquire a write lock, if the specified time elapses without the lock’s becoming available, an os_lock_timeout_exception exception is signaled. If the attempt causes a deadlock, the transaction is aborted regardless of the value of the specified timeout.

os_database::size()

os_unsigned_int32 size() const;

Returns the size in bytes of the specified database. If this number cannot be represented in 32 bits, err_misc is signaled.

os_database::size_in_sectors()

os_unsigned_int32 size_in_sectors() const;

Returns the size in sectors of the specified database. If this number cannot be represented in 32 bits, err_misc is signaled.

os_database::time_created()

os_unixtime_t time_created() const;

Returns the time at which the database was created.

os_database_reference

Instances of this type are stored in tables associated with cross-database pointers. Such pointers are resolved by database pathname.

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.

All ObjectStore programs must include the header file **<ostore/ostore.hh>**.

os_database_reference::os_database_reference()

os_database_reference(char *name);

Constructs a reference that stores the specified pathname. The **name** argument should be the pathname of the referent database as printed by the ObjectStore utility **ossize**, including the Server host prefix.

os_database_reference::~os_database_reference()

~os_database_reference();

Frees the memory associated with the specified **os_database_reference**'s name.

os_database_reference::delete_array()

```
static void delete_array(os_int32 n,
    os_database_referencep*array
);
```

Deletes an array returned by **os_database::get_database_references()**.

os_database_reference::get_name()

char *get_name();

Returns the pathname associated with the specified **os_database_reference**.

os_database_reference::operator ==()

os_boolean operator ==(os_database_reference const &dbref);

Returns nonzero if and only if the specified **os_database_ reference**s refer to the same database.

os_database_root

An object can be used as a database entry point if you associate a string with it by using a root, an instance of the system-supplied class **os_database_root**. Each root's sole purpose is to associate an object with a name. Once the association is made, you can retrieve a pointer to the object by performing a lookup on the name using a member function of the class **os_database**.

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.

All ObjectStore programs must include the header file **<ostore/ostore.hh>**.

os_database_root::~os_database_root()

~os_database_root();

Called when an instance of **os_database_root** is deleted. Deletes the associated name (persistent **char***) as well.

os_database_root::find()

static os_database_root *find(char *name, os_database *db);

Returns a pointer to the root in the specified database with the specified name. Returns **0** if not found.

os_database_root::get_name()

char *get_name();

Returns the name associated with the **os_database_root** for which the function is called.

os_database_root::get_typespec()

os_typespec *get_typespec();

Returns a pointer to the typespec associated with the **os_database_root** for which the function is called (the typespec last passed to **set_value()** for the root).

os_database_root::get_value()

void *get_value(os_typespec* = 0);

Returns a pointer to the entry-point object associated with the **os_database_root** for which the function is called. Note that the return value is typed as **void***, so a cast might be necessary when using it. If the specified **os_typespec** does not match the **os_typespec** specified when the value was set (see **os_database_root::set_value()** on page 104), err_pvar_type_mismatch is signaled. Note that this exception is signaled if and only if the specified typespec does not match the stored one; the actual type of the entry-point object is not checked.

os_database_root::set_value()

void set_value(void *new_value, os_typespec* = 0);

Establishes the object pointed to by **new_value** as the entry-point object associated with the **os_database_root** for which the function is called. If **new_value** points to transient memory or memory in a database other than the one containing the specified **os_database_root**, err_invalid_root_value is signaled. The specified **os_typespec** should designate the type of object pointed to by **new_value**. The typespec is stored for later use by **os_database_root::get_value()** (see above).

os_database_schema

This class is part of the ObjectStore metaobject protocol, which provides access to ObjectStore schemas. An instance of this class represents a database schema. **os_database_schema** is derived from **os_schema**.

Programs using this class must include **<ostore/ostore.hh>**, followed by **<ostore/coll.hh>** (if used), followed by **<ostore/mop.hh>**.

os_database_schema::get()

static const os_database_schema &get(const os_database&);

Returns the schema of the specified database. Signals err_no_schema if the specified database has no schema installed.

os_database_schema::get_for_update()

static os_database_schema &get_for_update(const os_database&);

Returns the schema of the specified database. Signals err_no_schema if the specified database has no schema installed. This differs from **get()** in that the return value is a reference to a non-**const os_database_schema** rather than a **const os_database_schema**.

os_database_schema::install()

void install(os_schema &new_schema);

Installs the classes in **new_schema** into the database schema specified by **this**.

os_dbutil

The database utility API provides C++ functions corresponding to the utilities documented in *ObjectStore Management*. You can use them as a basis for your own database utilities and tools.

All the functions in this facility are members of the class **os_dbutil**. Call the following function before using any other members of **os_dbutil**:

```
    static void os_dbutil::initialize();
```

You only need to call this function once in each application.

os_dbutil::chgrp()

```
static void chgrp(
    const char *pathname,
    const char *gname
);
```

Changes the primary group of the rawfs directory or database whose name is **pathname**. **gname** is the name of the new group.This function does not check for transaction consistency.

See **oschgrp** in *ObjectStore Management*.

os_dbutil::chmod()

```
static void chmod(
    const char *pathname,
    const os_unsigned_int32 mode
);
```

Changes the protections on the rawfs database or directory whose name is **pathname**. **mode** specifies the new protections. This function does not check for transaction consistency

See **oschmod** in *ObjectStore Management*.

os_dbutil::chown()

```
static void chown(
    const char *pathname,
    const char *uname
);
```

Changes the owner of the rawfs directory or database whose name is **pathname**. **uname** is the user name of the new owner. This function does not check for transaction consistency

See **oschown** in *ObjectStore Management*.

See Chapter 10, Database Utility API, in *ObjectStore Advanced C++ API User Guide*.

os_dbutil::close_all_server_connections()

static void close_all_server_connections();

Closes all connections the application has to ObjectStore Servers.

os_dbutil::close_server_connection()

static void close_server_connection(const char *hostname);

Closes the connection the application has to the ObjectStore Server running on the machine named **hostname**.

os_dbutil::cmgr_remove_file()

```
static char *cmgr_remove_file(
    const char *hostname,
    os_int32 cm_version_number
);
```

Makes the Cache Manager on the machine with the specified **hostname** delete all the cache and commseg files that are not in use by any client. The argument **cm_version_number** must match the Cache Manager's version number. Returns a pointer to the result message string.

See **oscmrf** in *ObjectStore Management*.

os_dbutil::cmgr_shutdown()

```
static char *cmgr_shutdown(
    const char *hostname,
    os_int32 cm_version_number
);
```

Shuts down the Cache Manager running on the machine with the specified **hostname**. The argument **cm_version_number** must match the Cache Manager's version number. Returns a pointer to the result message string.

See **oscmshtd** in *ObjectStore Management*.

os_dbutil::cmgr_stat()

```
static void cmgr_stat(
    const char *hostname,
    os_int32 cm_version_number,
    os_cmgr_stat *cmstat_data
);
```

Gets information for the Cache Manager on the machine with the specified **hostname**. The argument **cm_version_number** must match the Cache Manager's version number.

os_cmgr_stat public data members

cmstat_data points to an instance of **os_cmgr_stat** allocated by the caller, using the no-argument constructor. **os_cmgr_stat** has the following public data members:

os_unsigned_int32	**struct_version;**
os_unsigned_int32	**major_version;**
os_unsigned_int32	**minor_version;**
os_unsigned_int32	**pid;**
char	***executable_name;**
char	***host_name;**
os_unixtime_t	**start_time;**
os_int32	**soft_limit;**
os_int32	**hard_limit;**
os_int32	**free_allocated;**
os_int32	**used_allocated;**
os_int32	**n_clients;**
os_cmgr_stat_client	***per_client; /* array */**
os_int32	**n_servers;**
os_cmgr_stat_svr	***per_server; /* array */**
os_int32	**n_cache_file_usage;**
os_cmgr_stat_file_ usage	***cache_file_usage; /* array */**
os_int32	**n_comseg_file_usage;**
os_cmgr_stat_file_ usage	***comseg_file_usage; /* array */**
char	***cback_queue;**
char	***extra;**

The constructor sets **struct_version** to the value of **os_free_blocks_ version** in the **dbutil.hh** file included by your application. If this version is different from that used by the library, err_misc is signaled. The constructor initializes all other members to **0**.

Within the results returned by **os_dbutil::cmgr_stat**, the information about each client is provided in an object of class **os_cmgr_stat_client**. A data member has been added to class **os_cmgr_stat_client**, named **notification**, whose type is **os_cmgr_stat_notification***.

The value of this pointer is zero if the version of the Cache Manager does not support notifications (for example, precedes ObjectStore Release 4.0.2), or if this client is not using notifications because, for example, the client has not yet called **os_notification::subscribe** or **os_notification::_get_fd**.

Otherwise, the value is a pointer to an object of the class **os_cmgr_stat_notification**.

os_cmgr_stat_client public data members

os_cmgr_stat_client has the following public data members:

os_unsigned_int32	**struct_version;**
os_unsigned_int32	**queue_size;**
os_unsigned_int32	**n_queued;**
os_unsigned_int32	**n_received;**
os_unsigned_int32	**queue_overflows;**
os_unsigned_int32	**thread_state;**
os_int32	**pid;**
os_unsigned_int32	**euid;**
char*	**name;**
os_int32	**major_version;**
os_int32	**minor_version;**
os_int32	**commseg;**

os_cmgr_stat_svr public data members

os_cmgr_stat_svr has the following public data members:

os_unsigned_int32	**struct_version;**
char	***host_name;**
os_int32	**client_pid;**
char	***status_str;**

os_cmgr_stat_file_usage public data members

os_cmgr_stat_file_usage has the following public data members:

os_unsigned_int32	**struct_version;**

```
char                *file_name;
os_unsigned_int32   file_length;
os_boolean          is_free;
```

See **oscmstat** in *ObjectStore Management* for additional information.

os_dbutil::compare_schemas()

```
static os_boolean compare_schemas(
    const os_database* db1,
    const os_database* db2,
    os_boolean verbose = 1
);
```

Compares the schemas of **db1** and **db2**. Returns nonzero if the schemas are incompatible, **0** otherwise. Each database can contain an application schema, a compilation schema, or a database schema. If the database contains a database schema, it can be local or remote.

If **verbose** is nonzero, the function issues a message to the default output describing any incompatibility.

See **osscheq** in *ObjectStore Management*.

os_dbutil::copy_database()

```
static os_boolean copy_database(
    const char *src_database_name,
    const char *dest_database_name
);
```

Copies the database named **src_database_name** and names the copy **dest_database_name**. No transaction can be in progress. The source database cannot be in use. If there is already a database named **dest_database_name**, it is silently overwritten. Returns **0** for success, **1** if per-segment access control information has been changed during copy (which can happen when copying a rawfs database to a file database, since file databases do not have separate segment-level protections).

See **oscp** in *ObjectStore Management*.

os_dbutil::disk_free()

```
static void disk_free(
    const char *hostname,
    os_free_blocks *blocks
);
```

Gets disk space usage in the rawfs managed by the Server on the machine named **hostname**. **blocks** points to an instance of **os_free_blocks** allocated by the caller, using the zero-argument constructor.

os_free_blocks public data members

The class **os_free_blocks** has the following public data members:

os_unsigned_int32	**struct_version;**
os_unsigned_int32	**free_blocks;**
os_unsigned_int32	**file_system_size;**
os_unsigned_int32	**used_blocks;**

disk_free() sets the values of these data members for the instance of **os_free_blocks** pointed to by the argument **blocks**.

The **os_free_blocks** constructor sets **struct_version** to the value of **os_free_blocks_version** in the **dbutil.hh** file included by your application. If this version is different from that used by the library, err_misc is signaled.

See **osdf** in *ObjectStore Management*.

os_dbutil::expand_global()

```
static os_char_p *expand_global(
    char const *glob_path,
    os_unsigned_int32 &n_entries
);
```

Returns an array of pointers to rawfs pathnames matching **glob_path** by expanding **glob_path**'s wildcards (*****, **?**, **{}**, and **[]**). **n_entries** is set to refer to the number of pathnames returned. It is the caller's responsibility to delete the array and pathnames when they are no longer needed.

See **osglob** in *ObjectStore Management*.

os_dbutil::get_client_name()

static char const* get_client_name();

Returns the pointer last passed to **set_client_name()**. If there was no prior call to **set_client_name()**, **0** is returned. This function does not allocate any memory.

os_dbutil::get_sector_size()

static os_unsigned_int32 get_sector_size();

Returns 512, the size of a sector in bytes. Certain ObjectStore utilities report some of their results in numbers of sectors, and some Server parameters are specified in sectors. See *ObjectStore Management*.

os_dbutil::initialize()

static void initialize();

Call this before using any other members of **os_dbutil**.

os_dbutil::list_directory()

```
static os_rawfs_entry_p *list_directory(
    const char *path,
    os_unsigned_int32 &n_entries
);
```

Lists the contents of the rawfs directory named **path**. Returns an array of pointers to **os_rawfs_entry_p** objects. **n_entries** is set to the number of elements in the returned array. If **path** does not specify the location of a directory, err_not_a_directory is signaled. It is the caller's responsibility to delete the array and **os_rawfs_entry_p** objects when they are no longer needed.

See **osls** in *ObjectStore Management*.

os_dbutil::make_link()

```
static void make_link(
    const char *target_name,
    const char *link_name
);
```

Makes a rawfs soft link. **target_name** is the path pointed to by the link. **link_name** is the pathname of the link. Signals err_database_exists or err_directory_exists if **link_name** already points to an existing database or directory.

A rawfs can have symbolic links pointing within itself or to another Server's rawfs. The Server follows symbolic links within its rawfs for all **os_dbutil** members that pass pathname arguments, unless specified otherwise by a function's description; only **os_dbutil::stat()** can override this behavior. All members passing pathname arguments also follow cross-server links on the application side, unless specified otherwise by a function's description.

A rawfs symbolic link always has the ownership and the permissions of the parent directory.

See **osln**, **rehost_link**, and **rehost_all_links** in *ObjectStore Management*.

os_dbutil::mkdir()

```
static void mkdir(
    const char *path,
    const os_unsigned_int32 mode,
    os_boolean create_missing_dirs = 0
);
```

Makes a rawfs directory whose pathname is **path**. The new directory's protection is specified by **mode**. If **create_missing_dirs** is nonzero, creates the missing intermediate directories. Signals err_directory_not_found if **create_missing_dirs** is **0** and there are missing intermediate directories. Signals err_directory_exists if there is already a directory with the specified path. Signals err_database_exists if there is already a database with the specified path.

See **osmkdir** in *ObjectStore Management*.

os_dbutil::ossize()

```
static os_int32 ossize(
    const char *pathname,
    const os_size_options *options
);
```

Prints to standard output the size of the database whose name is **pathname**. **options** points to an instance of **os_size_options** allocated by the caller using the zero-argument constructor.

os_size_options public data members

os_size_options has the following public data members:

os_unsigned_int32	**struct_version;**
os_boolean	**flag_all; /* -a */**
os_boolean	**flag_segments; /* -c */**
os_boolean	**flag_total_database; /* -C */**
os_boolean	**flag_free_block_map; /* -f */**
os_unsigned_int32	**one_segment_number; /* -n */**

os_boolean	**flag_every_object; /* -o */**
char	**flag_summary_order;**
	/* -s 's'=space 'n'=number 't'=typename */
os_boolean	**flag_upgrade_rw; /* -u */**
os_boolean	**flag_internal_segments; /* -0 */**
os_boolean	**flag_access_control; /* -A */**

Each member corresponds to an option for the utility **ossize**. See **ossize** in *ObjectStore Management*.

The constructor sets **struct_version** to the value of **os_size_options_version** in the **dbutil.hh** file included by your application. If this version is different from that used by the library, err_misc is signaled. The constructor initializes all other members to **0**.

Returns **0** for success, **–1** for failure.

If **OS_ _DBUTIL_NO_MVCC** is set, this function opens the database for read only, rather than for MVCC (the default).

os_dbutil::osverifydb()

```
static os_unsigned_int32 osverifydb(
    const char *dbname,
    os_verifydb_options* opt= 0
);
```

Prints to standard output all pointers and references in the database named **dbname**. **opt** points to an instance of **os_verify_db_options** allocated by the caller using the zero-argument constructor. You must have called **os_collection::initialize()** and **os_mop::initialize()** prior to calling this function.

If **OS_ _DBUTIL_NO_MVCC** is set, this function opens the database for read only, rather than MVCC (the default).

os_verify_db_options public data members

os_verify_db_options has the following public data members:

os_boolean verify_segment_zero;	**/* verify the schema segment */**
os_boolean verify_collections;	**/* check all top-level collections */**
os_boolean verify_pointer_verbose;	**/* print pointers as they are verified */**
os_boolean verify_object_verbose;	**/* print objects as they are verified */**

```
os_boolean verify_references;          /* check all OS references */
os_int32 segment_error_limit;          /* maximum errors per segment */
os_boolean print_tag_on_errors;        /* print out the tag value on error */
const void* track_object_ptr;          /* Track object identified by pointer */
const char* track_object_ref_          /* Track the object identified by the
string;                                reference string. */
enum {
  default_action,
  ask_action,
  null_action,
} illegal_pointer_action;
```

Returns **0** for success, **1** for failure.

Upgrading databases created on 16K page size platforms

In releases prior to ObjectStore 5.1, there is a bug in our support of heterogeneous access to databases created on machines with a 16K page size: databases created on 16K page big-endian platforms cannot be accessed from small-endian platforms. Older databases might need to be upgraded. You can use the upgrade tool by means of the **os_verifydb_options** argument to the API **os_ dbutil::osverifydb()** with **os_verifydb_options::info_sector_tag_ verify_opt** set to the desired value:

```
class os_verifydb_options
{ public:
   ...
   enum info_sector_tag_verify_opt_enum {
      verify_skip = 0,            /* do not verify info sector tag */
      verify_report_only = 1,     /* report only */
      verify_upgrade = 2,         /* upgrade info sector tag */
      verify_skip_others = 4      /* skip other verifications */
   } info_sector_tag_verify_opt ;
   ...
}
```

Valid **os_verifydb_options::info_sector_tag_verify_opt** values are:

```
   verify_skip
   verify_report_only
   verify_upgrade
   verify_report_only | verify_skip_others
   verify_upgrade | verify_skip_others
```

See also, **osverifydb** in *ObjectStore Management*.

os_dbutil::osverifydb_one_segment()

```
static os_unsigned_int32 osverifydb_one_segment(
    const char *dbname,
    os_unsigned_int32 segment_number,
    os_unsigned_int32 starting_offset = 0,
    os_unsigned_int32 ending_offset = 0,
    os_verifydb_options *opt = 0
);
```

Prints to standard output all pointers and references in segment **segment_number** in the database named **dbname**. The offset items specify the starting and ending offsets (in bytes) within the segment where verification is done. If **starting_offset** is not specified, it defaults to **0**. This means verification starts at the beginning of the segment. Similarly, If **ending_offset** is not specified, it defaults to **0**. This means verify to the end of the segment. **opt** points to an instance of **os_verify_db_options** allocated by the caller using the zero-argument constructor.

os_verify_db_options public data members

os_verify_db_options has the following public data members:

```
os_boolean verify_segment_zero;      /* verify the schema segment */
os_boolean verify_collections;       /* check all top-level  collections */
os_boolean verify_pointer_verbose    /* print pointers as they are verified */
os_boolean verify_object_verbose     /* print objects as they are verified */
os_boolean verify_references;        /* check all OS references */
os_int32 segment_error_limit         /* maximum errors per segment */
os_boolean print_tag_on_errors       /* print out the tag value on error */
const void* track_object_ptr         /* Track object identified by pointer */
const char* track_object_ref_string  /* Track the object  identified by the
                                     reference string. */

enum {
  default_action,
  ask_action,
  null_action,
}
illegal_pointer_action;
```

Returns **0** for success, **1** for failure. You must have called **os_collection::initialize()** and **os_mop::initialize()** prior to calling this function. If **OS__DBUTIL_NO_MVCC** is set, this function opens the database for read only, rather than MVCC (the default).

See **osverifydb** in *ObjectStore Management*.

os_dbutil::osprmgc()

```
struct os_prmgc_options {
    os_boolean flag_quiet; // -q, default is false
    os_boolean flag_read_only; // -r, default is false
    os_boolean flag_one_segment; // -n, default is false
    os_unsigned_int32 one_segment_number; // the N in -n N
    os_prmgc_type prmgc_type; //-t default is remove_whole_ranges
};
```

This is particularly useful for databases with discriminant unions.

Discriminant union considerations

If you have a database with discriminant unions, you must perform PRM garbage collection using **os_dbutil::osprmgc**, and link in the necessary discriminant functions.

Both the command-line and embedded versions of the utility use a streaming fetch policy. The embedded version ensures that the policy is restored to its original state (if different) to minimize impact on the application. See osprmgc: Trimming Persistent Relocation Maps in *ObjectStore Management* for further detail.

os_dbutil::rehost_all_links()

```
static void rehost_all_links(
    const char *server_host,
    const char *old_host,
    const char *new_host
);
```

Changes the hosts of specified rawfs links. **server_host** specifies the host containing the links to be changed. All links pointing to **old_host** are changed to point to **new_host**. On some operating systems, you must have special privileges to use this function.

See **oschhost** in *ObjectStore Management*.

os_dbutil::rehost_link()

```
static void rehost_link(
    const char *pathname,
    const char *new_host
);
```

Changes the host to which a rawfs link points. **pathname** must specify a symbolic link, otherwise err_not_a_link is signaled. **new_host** specifies the new Server host.

See **oschhost** in *ObjectStore Management*.

os_dbutil::remove()

static void remove(char const *path);

Removes the database or rawfs link with the specified name. If **path** names a link, the link is removed but the target of the link is not. If **path** names a directory, it is not removed. Signals err_not_a_database if **path** exists in a rawfs but is not a database. Signals err_file_error when a problem is reported by the Server host's file system. Signals err_file_not_local if the file is not local to this Server.

See **osrm** in *ObjectStore Management*.

os_dbutil::rename()

```
static void rename(
    const char *source,
    const char *target
);
```

Renames the rawfs database or directory. **source** is the old name and **target** is the new name. Signals err_cross_server_rename if **source** and **target** are on different Servers. Signals err_invalid_rename if the operation makes a directory its own descendent. Signals err_database_exists if a database named **target** already exists. Signals err_directory_exists if a directory named **target** already exists.

See **osmv** in *ObjectStore Management*.

os_dbutil::rmdir()

static void rmdir(const char *path);

Removes the rawfs directory with the specified pathname. Signals err_directory_not_empty if the directory still contains databases. Signals err_not_a_directory if the argument does not specify a directory path.

See **osrmdir** in *ObjectStore Management*.

os_dbutil::set_application_schema_path()

```
static char *set_application_schema_path(
    const char *executable_pathname,
    const char *database_pathname
);
```

Finds or sets an executable's application schema database. **executable_pathname** specifies the executable. **database_pathname** is either the new schema's pathname or **0**. If **database_pathname** is **0**, the function returns new storage containing the current pathname. If **database_pathname** is nonzero, the function returns **0**.

See **ossetasp** in *ObjectStore Management*.

os_dbutil::set_client_name()

```
static void set_client_name(const char *name);
```

Sets the client name string for message printing.

os_dbutil::stat()

```
static os_rawfs_entry *stat(
    const char *path,
    const os_boolean b_chase_links = 1
);
```

Gets information about a rawfs pathname. If **b_chase_links** is false and the path is a link, the Server does not follow it. The Server still follows intrarawfs links for the intermediate parts of the path.

Returns a pointer to an **os_rawfs_entry** to be destroyed by the caller, or **0** on error.

See **ostest** and **osls** in *ObjectStore Management*.

os_dbutil::svr_checkpoint()

```
static os_boolean svr_checkpoint(
    const char *hostname
);
```

Makes the specified Server take a checkpoint asynchronously. Returns nonzero when successful, **0** or an exception on failure.

See **ossvrchkpt** in *ObjectStore Management*.

os_dbutil::svr_client_kill()

```
static os_boolean svr_client_kill(
    const char *server_host,
    os_int32 client_pid,
    const char *client_name,
    const char *client_hostname,
    os_boolean all,
    os_int32 &status
);
```

Kills one or all clients of the specified Server. **hostname** is the name of the machine running the Server.

client_pid is the process ID of the client to kill. **client_name** is the name of the client to kill. **client_hostname** is the name of the machine running the client to kill.

If **all** is **1**, the other arguments except **server_host** are ignored, and all clients on the specified Server are killed.

status is set to **–2** if a client was killed, **0** if no clients matched the specifications, **2** if multiple clients matched the specification. Any other value means that access was denied.

Returns **0** for failure and nonzero for success.

See **ossvrclntkill** in *ObjectStore Management*.

os_dbutil::svr_ping()

```
static char *svr_ping(
    const char *server_host,
    os_svr_ping_state &state
);
```

Determines whether an ObjectStore Server is running on the machine named **server_host**. The referent of **state** is set to one of the following global enumerators:

- **os_svr_ping_is_alive**
- **os_svr_ping_not_reachable**
- **os_svr_ping_no_such_host**

Returns a pointer to the status message string.

Failover servers

os_dbutil::svr_ping() returns the enumerator **os_svr_ping_is_ alive_as_failover_backup** if it is possible that the Server is running in backup failover mode. The heuristic for determining if it possibly is running in backup mode is if the exception err_broken_ connection is raised while pinging a Server and the current locator file indicates that the Server is a member of a failover pair.

See **ossvrping** in *ObjectStore Management*.

os_dbutil::svr_shutdown()

```
static os_boolean svr_shutdown(
    const char *server_host
);
```

Shuts down the Server on the machine named **server_host**. Returns nonzero for success, **0** otherwise. On some operating systems, you must have special privileges to use this function.

See **ossvrshtd** in *ObjectStore Management*.

os_dbutil::svr_stat()

```
static void svr_stat(
    const char *server_host,
    os_unsigned_int32 request_bits,
    os_svr_stat *svrstat_data
);
```

Gets statistics for a Server's clients on **server_host**.

request_bits specifies what information is desired. Supply this argument by forming the bit-wise disjunction of zero or more of the following enumerators:

- **os_svr_stat::get_svr_usage**
- **os_svr_stat::get_svr_meter_samples**
- **os_svr_stat::get_svr_parameters**
- **os_svr_stat::get_svr_parameters**
- **os_svr_stat::get_client_info_others**

For each enumerator that is specified, the corresponding information is retrieved.

For each of the classes described below, the constructor sets **struct_version** to the value of **os_free_blocks_version** in the **dbutil.hh** file included by your application. If this version is different from that used by the library, err_misc is signaled. The constructor initializes all other members to **0**.

```
    os_dbutil::svr_stat(db->get_host_name(),
    os_svr_stat::get_svr_meter_samples,&svrstat)
    n_sent = svrstat.svr_meter_samples->n_notifies_sent;
n_received = svrstat.svr_meter_samples->n_notifies_ received;
```

svrstat_data public data members

svrstat_data points to an instance of **os_svr_stat** allocated by the caller, using the zero-argument constructor. This structure has the following public data members:

os_unsigned_int32	**struct_version;**
os_svr_stat_svr_header	**header;**
os_svr_stat_svr_parameters*	**svr_parameters;**
os_svr_stat_svr_rusage*	**svr_rusage;**
os_svr_stat_svr_meters*	**svr_meter_samples;**
os_unsigned_int32	**n_meter_samples;**
os_svr_stat_client_info*	**client_info_self;**
os_svr_stat_client_info*	**client_info_others;**
os_unsigned_int32	**n_clients;**

Set the pointer-valued members to point to classes you allocate using zero-argument constructors.

os_svr_stat_svr_header public data members

os_svr_stat_svr_header has the following public data members:

os_unsigned_int32	**struct_version;**
char*	**os_release_name;**
os_unsigned_int32	**server_major_version;**
os_unsigned_int32	**server_minor_version;**
char*	**compilation;**

os_svr_stat_svr_parameters public data members

os_svr_stat_svr_parameters has the following public data members:

os_unsigned_int32	**struct_version;**
char*	**parameter_file;**
os_boolean	**allow_shared_mem_usage;**
os_int32*	**authentication_list;**
os_unsigned_int32	**n_authentications;**
os_int32	**RAWFS_db_expiration_time;**
os_int32	**deadlock_strategy;**
os_unsigned_int32	**direct_to_segment_threshold;**
char*	**log_path;**
os_unsigned_int32	**current_log_size_sectors;**

```
os_unsigned_int32    initial_log_data_sectors;
os_unsigned_int32    growth_log_data_sectors;
os_unsigned_int32    log_buffer_sectors;
os_unsigned_int32    initial_log_record_sectors;
os_unsigned_int32    growth_log_record_sectors;
os_unsigned_int32    max_data_propagation_threshold;
os_unsigned_int32    max_propagation_sectors;
os_unsigned_int32    max_msg_buffer_sectors;
os_unsigned_int32    max_msg_buffers;
os_unsigned_int32    sleep_time_between_2p_outcomes;
os_unsigned_int32    sleep_time_between_propagates;
os_unsigned_int32    write_buffer_sectors;
os_unsigned_int32    tcp_recv_buffer_size;
os_unsigned_int32    tcp_send_buffer_size;
os_boolean           allow_nfs_locks;
os_boolean           allow_remote_database_access;
os_unsigned_int32    max_two_phase_delay;
os_unsigned_int32    max_aio_threads;
os_unsigned_int32    cache_mgr_ping_time;
os_unsigned_int32    max_memory_usage;
os_unsigned_int32    max_connect_memory_usage;
os_unsigned_int32    remote_db_grow_reserve
os_boolean           allow_estale_to corruptDBs
os_int32             restricted_file_db_access_only
os_unsigned_int32    failover_heartbeat_time
```

os_svr_stat_svr_ rusage public data members

os_svr_stat_svr_rusage has the following public data members:

```
os_unsigned_int32          struct_version;
os_timesecs ru_utime;      /* user time used */
os_timesecs ru_stime;      /* system time used */
os_int32     ru_maxrss;    /* maximum resident set size */
os_int32     ru_ixrss;     /* integral shared memory size */
os_int32     ru_idrss;     /* integral unshared data size */
os_int32     ru_isrss;     /* integral unshared stack size */
os_int32     ru_minflt;    /* page reclaims */
os_int32     ru_majflt;    /* page faults */
```

```
os_int32   ru_nswap;     /* swaps */
os_int32   ru_inblock;   /* block input operations */
os_int32   ru_oublock;   /* block output operations */
os_int32   ru_msgsnd;    /* messages sent */
os_int32   ru_msgrcv;    /* messages received */
os_int32   ru_nsignals;  /* signals received */
os_int32   ru_nvcsw;     /* voluntary context switches */
os_int32   ru_nivcsw;    /* involuntary context switches */
```

The classes **os_svr_stat_svr_meters** and **os_svr_stat_client_meters** each have data members named **n_notifies_sent** and **n_notifies_received**. All four are of type **os_unsigned_int32**. They give the total number of notifications that have been received by the Server, and the number that have been sent by the Server, giving the total for the Server as a whole, and the total for each client.

os_svr_stat_svr_meters public data members

os_svr_stat_svr_meters has the following public data members:

```
os_unsigned_int32     struct_version;
os_boolean            valid;
os_unsigned_int32     n_minutes;
os_unsigned_int32     n_receive_msgs;
os_unsigned_int32     n_callback_msgs;
os_unsigned_int32     n_callback_sectors_requested;
os_unsigned_int32     n_callback_sectors_succeeded;
os_unsigned_int32     n_sectors_read;
os_unsigned_int32     n_sectors_written;
os_unsigned_int32     n_commit;
os_unsigned_int32     n_phase_2_commit;
os_unsigned_int32     n_readonly_commit;
os_unsigned_int32     n_abort;
os_unsigned_int32     n_phase_2_abort;
os_unsigned_int32     n_deadlocks;
os_unsigned_int32     n_msg_buffer_waits;
os_unsigned_int32     n_log_records;
os_unsigned_int32     n_log_seg_switches;
os_unsigned_int32     n_flush_log_data_writes;
os_unsigned_int32     n_flush_log_record_writes;
```

os_unsigned_int32	**n_log_data_writes;**
os_unsigned_int32	**n_log_record_writes;**
os_unsigned_int32	**n_sectors_propagated;**
os_unsigned_int32	**n_sectors_direct;**
os_unsigned_int32	**n_do_some_propagation;**
os_unsigned_int32	**n_notifies_sent**
os_unsigned_int32	**n_notifies_received**
os_unsigned_int32	**n_lock_waits**
os_unsigned_int32	**total_lock_wait_times**

os_svr_stat_client_info public data members

os_svr_stat_client_info has the following public data members:

os_unsigned_int32	**struct_version;**
os_svr_stat_client_process*	**process;**
os_svr_stat_client_state*	**state;**
os_svr_stat_client_meters*	**meters;**

Set the pointer-valued members to point to classes you allocate using zero-argument constructors.

os_svr_stat_client_process public data members

os_svr_stat_client_process has the following public data members:

os_unsigned_int32	**struct_version;**
char*	**host_name;**
os_unsigned_int32	**process_id;**
char*	**client_name;**
os_unsigned_int32	**client_id;**

os_svr_stat_client_state public data members

os_svr_stat_client_state has the following public data members:

os_unsigned_int32	**struct_version;**
os_client_state_type	**client_state;**
char*	**message_name;**
os_boolean	**txn_in_progress;**
os_unsigned_int32	**txn_priority;**
os_unsigned_int32	**txn_duration;**
os_unsigned_int32	**txn_work;**
os_client_lock_type	**lock_state;**
os_unsigned_int32	**db_id;**
char*	**db_pathname;**

os_unsigned_int32	**locked_seg_id;**
os_unsigned_int32	**locking_start_sector;**
os_unsigned_int32	**locking_for_n_sectors;**
os_unsigned_int32	**n_conflicts;**
os_svr_stat_client_ process*	**lock_conflicts;**

```
enum os_client_lock_type {
  OSSVRSTAT_CLIENT_LOCK_TO_MAX_BLOCKS,
  OSSVRSTAT_CLIENT_LOCK_TO_NBLOCKS,
};
enum os_client_state_type {
  OSSVRSTAT_CLIENT_WAITING_MESSAGE,
  OSSVRSTAT_CLIENT_EXECUTING_MESSAGE,
  OSSVRSTAT_CLIENT_WAITING_RANGE_READ_LOCK,
  OSSVRSTAT_CLIENT_WAITING_RANGE_WRITE_LOCK,
  OSSVRSTAT_CLIENT_WAITING_SEGMENT_WRITE_LOCK,
  OSSVRSTAT_CLIENT_DEAD,
  OSSVRSTAT_CLIENT_WAITING_SEGMENT_READ_LOCK,
  OSSVRSTAT_CLIENT_WAITING_DB_READ_LOCK,
  OSSVRSTAT_CLIENT_WAITING_DB_WRITE_LOCK,
};
```

Data in **locking_start_sector** and **locking_for_n_sectors** is valid only when **lock_state** is **OSSVRSTAT_CLIENT_STATE_WAITING_ RANGE_READ_LOCK** or **OSSVRSTAT_CLIENT_STATE_WAITING_ RANGE_WRITE_LOCK**.

os_svr_stat_client_ meters public data members

os_svr_stat_client_meters has the following public data members:

os_unsigned_int32	**struct_version;**
os_unsigned_int32	**n_receive_msgs;**
os_unsigned_int32	**n_callback_msgs;**
os_unsigned_int32	**n_callback_sectors_requested;**
os_unsigned_int32	**n_callback_sectors_succeeded;**
os_unsigned_int32	**n_sectors_read;**
os_unsigned_int32	**n_sectors_written;**
os_unsigned_int32	**n_deadlocks;**
os_unsigned_int32	**n_lock_timeouts;**

```
os_unsigned_int32        n_commit;
os_unsigned_int32        n_phase_2_commit;
os_unsigned_int32        n_readonly_commit;
os_unsigned_int32        n_abort;
os_unsigned_int32        n_phase_2_abort;
os_unsigned_int32        n_notifies_sent
os_unsigned_int32        n_notifies_received
```

See **ossvrstat** and **ossvrmtr** in *ObjectStore Management*.

os_DLL_finder

```
class os_DLL_finder {
public:
```

Functions in this class are used to create a DLL identifier prefix. Create a subclass of this class to implement a new kind of DLL identifier. A prefix before a colon in a DLL identifier string maps to a DLL finder subclass.

Required header files **<ostore/client/dll_fndr.hh>**

os_DLL_finder::register_()

```
void register_(const char* prefix);
```

Register this as the finder for DLL identifiers with the given **prefix**.

os_DLL_finder::unregister()

```
void unregister(const char* prefix);
```

Unregisters a DLL finder sto that it is no longer the finder for DLL identifiers with the given **prefix**.

os_DLL_finder::get()

```
static os_DLL_finder* get(const char* DLL_identifier);
```

Gets the finder for the specified **DLL_identifier**'s prefix or returns null if none is registered.

os_DLL_finder::equal_DLL_identifiers()

```
static os_boolean equal_DLL_identifiers(
    const char* id1, const char* id2
);
```

Compares two DLL identifier strings and returns **true** if the identifier strings are eqivalent.

os_DLL_finder::load_DLL

```
virtual os_DLL_handle load_DLL(
    const char* DLL_identifier,
    os_boolean error_if_not_found) = 0;
```

Each subclass of **os_DLL_finder** must provide an implementation of **load_DLL()** that interprets the suffix part of the DLL identifier and calls the appropriate operating system API (or calls another **os_DLL_finder**) to load the DLL.

os_DLL_finder::load_DLL

```
virtual os_boolean equal_DLL_identifiers_same_prefix(
    const char* id1,
    const char* id2) = 0;
```

Each subclass of **os_DLL_finder** must provide an implementation of **equal_DLL_identifiers_same_prefix** that compares two DLL identifiers that are both implemented by this finder and returns true if they are equal.

os_DLL_schema_info

This class provides access to information in a DLL about its DLL schema, including the pathname of the schema database, DLL identifiers, rep descriptors, pointers to vtbls, and so on. Its base class is **os_schema_info.**

Required header files **#include <ostore/nreloc/schftyps.hh>**

os_DLL_schema_info::add_DLL_identifier()

void add_DLL_identifier(const char* id);

Adds the specified DLL identifier to the set of DLL identifiers of the **os_DLL_schema_info**. This function can be called before **os_DLL_schema_info::DLL_loaded()** if the DLL identifier is determined independently of the schema.

This can be used, for example, in a case where one developer hands off the schema database, schema file, and other object files to another developer who incorporates these into a DLL that he builds. The schema starts with a dummy DLL identifier to make it a DLL schema, and has the real identifiers added by the second developer.

The **add_DLL_identifier()** function allocates a small amount of memory that is never freed. The amount of memory is proportional to the total number of DLL identifiers in the **os_DLL_schema_info**. It also retains a pointer to the **id** argument indefinitely.

os_DLL_schema_info::DLL_loaded()

os_schema_handle& DLL_loaded();

Notifies ObjectStore that a DLL has been loaded and that the DLL's schema must be loaded and merged into the process's complete program schema. The **this** argument identifies the schema to be loaded. Typically the **this** argument is an **os_DLL_schema_info** structure generated by **ossg** in the DLL that is calling **DLL_loaded()**. Typically the call is in the DLL's initialization function.

Upon notification, one of the following then occurs:

- If ObjectStore is not yet initialized or there is no current transaction, this function saves the arguments for later processing when the first database is jput in use by the next transaction. The arguments are saved in the form of an **os_ schema_handle** object that is not fully initialized.
- If the DLL schema is already loaded, this function returns its **os_schema_handle**.
- For all other cases, **os_DLL_schema_info::DLL_loaded()** finds the schema and rep descriptors and loads them.

Aborting a transaction does not roll back **os_DLL_schema_ info::DLL_loaded()**.

The returned **os_schema_handle** represents the DLL schema that was or will be loaded.

Debugging

Delayed loading of DLL schema after calling **os_DLL_schema_ info::DLL_loaded()** can raise exceptions. It can be difficult to debug these, because they occur later than the program action that loaded the DLL, but the error message should always include a DLL identifier of the DLL.

One way to debug such a program is to start a transaction and put a database in use to force deferred loading to happen.

```
os_schema_handle&  DLL_loaded(
    const char* explicit_schema_database_path
);
```

The **explicit_schema_database_path** argument allows the file pathname of the DLL schema database to be passed in, overriding the pathname in the **os_DLL_schema_info**. This calls **os_DLL_ schema_info::set_schema_database_pathname** and then calls the no-arguments overloading of **DLL_loaded** described previously.

os_DLL_schema_info::DLL_unloaded()

```
void DLL_unloaded();
```

Uses the **os_DLL_schema_info** to locate a loaded **os_schema_ handle** and calls **os_schema_handle::DLL_unloaded()**.

An exception is thrown if no corresponding **os_schema_handle** currently exists.

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.

By default, **os_dynamic_extent** does not search subclasses when the requested object type is a class type. To enable this behavior, set the argument **include_subclasses** to true. When this behavior is enabled, **os_dynamic_extent** searches all classes that the requested class type is derived from.

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,
os_boolean include_subclasses=0
);
```

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**.

By default, **os_dynamic_extent** does not search subclasses when the requested object type is a class type. Set the argument **include_subclasses** to true to enable **os_dynamic_extent** to search all classes that the requested class type is derived from.

```
os_dynamic_extent(
    os_typespec * typespec,
    os_boolean options = os_dynamic_extent::all_segments
    os_boolean include_subclasses=0
);
```

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.

By default, **os_dynamic_extent** does not search subclasses when the requested object type is a class type. Set the argument **include_subclasses** to true to enable **os_dynamic_extent** to search all classes that the requested class type is derived from.

```
os_dynamic_extent(
    os_database * db,
    os_typespec* typespec,
    os_segment* seg
    os_boolean include_subclasses=0
);
```

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.**

By default, **os_dynamic_extent** does not search subclasses when the requested object type is a class type. Set the argument **include_subclasses** to true to enable **os_dynamic_extent** to search all classes that the requested class type is derived from.

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; **0** is returned otherwise.

os_dynamic_extent::~os_dynamic_extent()

~os_dynamic_extent();

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

os_enum_type

class os_enum_type : public os_type

This class is part of the ObjectStore metaobject protocol, which provides access to ObjectStore schemas. An instance of this class represents a C++ enumeration type. This class is derived from **os_ type**.

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.

Programs using this class must include **<ostore/ostore.hh>**, followed by **<ostore/coll.hh>** (if used), followed by **<ostore/mop.hh>**.

os_enum_type::create()

```
static os_enum_type &create(
    const char *name,
    os_List<os_enumerator_literal*>&
);
```

Creates an instance of **os_enum_type** named **name**. The specified list contains the enumerators of the created enumeration type.

os_enum_type::get_name()

const char *get_name() const;

Returns the name of the specified enumeration. A zero-length string is returned for an anonymous enumeration.

os_enum_type::get_enumerator()

const os_enumerator_literal *get_enumerator(os_int32) const;

Returns the enumerator that names the specified integer. Returns **0** if there is no enumerator with the specified value. If there is more than one enumerator with the specified value, the first one is returned.

os_enum_type::get_enumerators()

os_List<const os_enumerator_literal*> get_enumerators() const;

Returns a list, in declaration order, of the enumerator literals defined by the enumeration.

os_enum_type::get_pragmas()

```
os_List<os_pragma*> get_pragmas() const;
```

Returns the pragmas associated with the specified enumeration.

os_enum_type::get_source_position()

```
void get_source_position (
    const char* &file,
    os_unsigned_int32 &line
) const;
```

Returns the source position associated with the specified enumeration.

os_enum_type::set_enumerators()

```
void set_enumerators(os_List<os_enumerator_literal*>&);
```

Specifies, in declaration order, the enumerator literals defined by the specified enumeration.

os_enum_type::set_name()

```
void set_name(const char *);
```

Sets the name of the specified enumeration.

os_enum_type::set_pragmas()

```
void set_pragmas(os_List<os_pragma*>);
```

Sets the pragmas associated with the specified enumeration.

os_enum_type::set_source_position()

```
void set_source_position(
    const char* file,
    os_unsigned_int32 line
);
```

Sets the source position associated with the specified enumeration.

os_enumerator_literal

This class is part of the ObjectStore metaobject protocol, which provides access to ObjectStore schemas. An instance of this class represents a C++ enumerator.

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_enumerator_literal::create()

static os_enumerator_literal& create(const char*, os_int32);

Creates an **os_enumerator_literal** of the specified name and value.

os_enumerator_literal::set name()

void set_name(const char*);

Sets the name of the specified enumerator.

os_enumerator_literal::set_value()

void set_value(os_int32);

Sets the integer value of the specified enumerator.

os_enumerator_literal::get_name()

const char *get_name() const;

Returns the name of the specified enumerator.

os_enumerator_literal::get_value()

os_int32 get_value() const;

Returns the integer value of the specified enumerator.

os_evolve_subtype_fun_binding

Instances of this class represent an association between a class and a reclassifier function. Instances of **os_evolve_subtype_fun_binding** are used as arguments to **os_schema_evolution::augment_subtype_selectors()**. Instances should be allocated in transient memory only.

Programs using this class must include **<ostore/ostore.hh>,** followed by **<ostore/coll.hh>** (if used), followed by **<ostore/schmevol.hh>**.

os_evolve_subtype_fun_binding::os_evolve_subtype_fun_binding()

```
os_evolve_subtype_fun_binding(
    char *class_name,
    char* (*f)(const os_typed_pointer_void&)
);
```

Associates the class named **class_name** with the function **f**.

os_failover_server

class os_failover_server : public os_server

This class is derived from **os_server**.

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.

Programs using this class must include **<ostore/ostore.hh>**, followed by **<ostore/coll.hh>** (if collections are used).

See *ObjectStore Management* for more information on failover.

os_failover_server::get_logical_server_hostname()

char* get_logical_server_hostname() const;

Returns the logical name of a failover server. A failover server should always be referred to by its logical server name.

The caller should delete the returned value.

os_failover_server::get_online_server_hostname()

char* get_online_server_hostname() const;

Returns the name of the Server that the client is currently connected to, either the logical server, alternative server, or the empty string if there is no connection.

The caller should delete the returned value.

os_failover_server::get_reconnect_retry_interval()

os_unsigned_int32 get_reconnect_retry_interval() const;

Returns the reconnect retry interval, which determines how often to ping the Servers of a failover Server pair while attempting to reconnect to them.

os_failover_server::get_reconnect_timeout()

os_unsigned_int32 get_reconnect_timeout() const;

Returns the maximum amount of time that a client application attempts to reconnect to a broken failover Server connection.

When the timeout is reached, the following exception is raised: err_broken_failover_server_connection

os_failover_server::set_reconnect_timeout_and_interval()

```
os_boolean set_reconnect_timeout_and_interval(
    os_unsigned_int32 total_timeout_secs,
    os_unsigned_int32 interval_secs);
```

Sets the total amount of time to try to reconnect a broken connection to a failover Server. The **interval_secs** argument is used to control how frequently the Servers of a failover Server pair are pinged to see if they are available.

Returns true if the function has reset these parameters with the given argument values.

If the parameters are invalid, the function returns false and does not change the reconnect timeout or reconnect retry interval.

Invalid parameters are those for which

- **interval_secs** is greater than **total_timeout_secs**.
- **interval_secs** is set to **0** when **total_timeout_secs** is nonzero.

os_field_member_variable

class os_field_member_variable : public os_member_variable

This class is part of the ObjectStore metaobject protocol, which provides access to ObjectStore schemas. An instance of this class represents a bit-field member. This class is derived from **os_member_variable**.

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_int8** is defined as an unsigned 8-bit integer type.

Programs using this class must include **<ostore/ostore.hh>**, followed by **<ostore/coll.hh>** (if used), followed by **<ostore/mop.hh>**.

os_field_member_variable::create()

```
static os_field_member_variable& create(
    const char* name, os_type* type,
    os_unsigned_int8 size_in_bits
);
```

Creates an **os_field_member_variable** of the specified **name**, **type**, and **size_in_bits**.

os_field_member_variable::get_offset()

```
void get_offset(
    os_unsigned_int32 &bytes, os_unsigned_int8 &bits
) const;
```

Returns the offset in bytes and bits to the specified bit field.

os_field_member_variable::get_size()

```
os_unsigned_int8 get_size() const;
```

Returns the size in bits of the specified bit field.

os_field_member_variable::set_size()

```
void set_size(os_unsigned_int8 size_in_bits);
```

Sets the size in bits of the specified bit field.

os_function_type

class os_function_type : public os_type

This class is part of the ObjectStore metaobject protocol, which provides access to ObjectStore schemas. An instance of this class represents a C++ function type. This class is derived from **os_type**.

Programs using this class must include **<ostore/ostore.hh>**, followed by **<ostore/coll.hh>** (if used), followed by **<ostore/mop.hh>**.

os_function_type::create()

```
static os_function_type &create(
    os_arg_list_kind arg_list_kind,
    os_List<os_type*> &args,
    os_type *return_type
);
```

Creates an **os_function_type**. The type of the new function's n^{th} argument is the n^{th} element of **args**. The return type is **return_type**. The possible values of **arg_list_kind** are **os_function_ type::Unknown**, **os_function_type::Variable**, and **os_function_ type::Known**. See **os_function_type::get_arg_list_kind()** on page 143.

os_function_type::equal_signature()

```
os_boolean equal_signature(
    const os_function_type &other_func,
    os_boolean check_return_type = 0
) const;
```

Returns nonzero if the specified function types are equivalent. If **check_return_type** is **0**, returns nonzero if the arguments are the same.

os_function_type::get_arg_list()

```
os_list get_arg_list() const;
```

Returns a list of **os_type***s, pointers to the argument types of the specified function type. See also the member **get_arg_list_kind()** below.

os_function_type::get_arg_list_kind()

enum os_arg_list_kind { Unknown, Known, Variable } ;

os_arg_list_kind get_arg_list_kind() const;

Returns an enumerator indicating the type of argument list associated with the specified function. **os_function_ type::Unknown** indicates that the argument profile is unknown; a call to **os_function_type::get_arg_list()** will return an empty list. **os_function_type::Variable** indicates that the function accepts a variable number of arguments; a call to **os_function_type::get_ arg_list()** will return the list of known leading arguments. **os_ function_type::Known** indicates that the function takes a known fixed number of arguments; a call to **os_function_type::get_arg_ list()** will return the complete argument list.

os_function_type::get_return_type()

os_type &get_return_type() const;

Returns the return type associated with the specified function.

os_function_type::set_arg_list()

void set_arg_list(os_List<os_type*>);

Returns a list of **os_type***s, pointers to the argument types of the specified function type. See also the member **get_arg_list_kind()**, above.

os_function_type::set_arg_list_kind()

void set_arg_list_kind(os_int32);

Specifies an enumerator indicating the type of argument list associated with the specified function. **os_function_ type::Unknown** indicates that the argument profile is unknown. **os_function_type::Variable** indicates that the function accepts a variable number of arguments. **os_function_type::Known** indicates that the function takes a known fixed number of arguments.

os_function_type::set_return_type()

void set_return_type(os_type &);

Sets the return type associated with the specified function.

os_indirect_type

class os_indirect_type : public os_type

This class is part of the ObjectStore metaobject protocol, which provides access to ObjectStore schemas. An instance of this class is either an **os_named_indirect_type** (typedef) or **os_anonymous_indirect_type** (a type with **const** or **volatile** specifiers). This class is derived from **os_class_type**.

os_indirect_type::get_target_type()

const os_type &get_target_type() const;

Returns the type for which **this** is a typedef or which **this** qualifies with a **const** or **volatile** specifier.

os_type &get_target_type();

Returns the type for which **this** is a typedef or which **this** qualifies with a **const** or **volatile** specifier.

os_indirect_type::set_target_type()

void set_target_type(os_type &);

Sets the type for which **this** is a typedef or which **this** qualifies with a **const** or **volatile** specifier.

os_instantiated_class_type

class os_instantiated_class_type : public os_class_type

This class is part of the ObjectStore metaobject protocol, which provides access to ObjectStore schemas. An instance of this class represents an instantiation of a template class. This class is derived from **os_class_type**.

os_instantiated_class_type::create()

static os_instantiated_class_type& create(const char* name);

Creates an **os_instantiated_class_type** with the specified name.

```
static os_instantiated_class_type& create(
    const char* name,
    os_List<os_base_class*>&,
    os_List<os_member*>&,
    os_template_instantiation*,
    os_boolean defines_virtual_functions
);
```

Creates an **os_instantiated_class_type** from the specified template instantiation and with the specified name, base classes, and members.

os_instantiated_class_type::get_instantiation()

const os_template_instantiation &get_instantiation() const;

Returns a reference to a **const os_template_instantiation** that represents the template instantiation resulting in this class.

os_template_instantiation &get_instantiation();

Returns a reference to a non-**const os_template_instantiation** that represents the template instantiation resulting in this class.

os_instantiated_class_type::set_instantiation()

void set_instantiation(os_template_instantiation&);

Sets the **os_template_instantiation** for the specified **os_ instantiated_class_type**.

os_integral_type

class os_integral_type : public os_type

This class is part of the ObjectStore metaobject protocol, which provides access to ObjectStore schemas. An instance of this class represents a C++ integer type. This class is derived from **os_type**. Performing **os_type::kind()** on an **os_integral_type** returns one of the following enumerators: **os_type::Signed_char**, **os_ type::Unsigned_char**, **os_type::Signed_short**, **os_type::Unsigned_ short**, **os_type::Integer**, **os_type::Unsigned_integer**, **os_ type::Signed_long**, or **os_type::Unsigned_long**.

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_integral_type::create()

static os_integral_type &create(const char*);

Creates an **os_integral_type** representing the type with the specified name.

os_integral_type::create_defaulted_char()

static os_integral_type &create_defaulted_char(os_boolean signed);

Creates an **os_integral_type** representing the type **char**.

os_integral_type::create_int()

static os_integral_type &create_int(os_boolean signed);

Creates an **os_integral_type** representing the type **int** or **unsigned int**.

os_integral_type::create_long()

static os_integral_type &create_long(os_boolean signed);

Creates an **os_integral_type** representing the type **long** or **unsigned long**.

os_integral_type::create_short()

static os_integral_type &create_short(os_boolean signed);

Creates an **os_integral_type** representing the type **short** or **unsigned short**.

os_integral_type::create_signed_char()

static os_integral_type &create_signed_char();

Creates an **os_integral_type** representing the type **signed char**.

os_integral_type::create_unsigned_char()

static os_integral_type &create_unsigned_char();

Creates an **os_integral_type** representing the type **unsigned char**.

os_integral_type::is_signed()

os_boolean is_signed() const;

Returns **1** if and only if the specified object represents a signed type.

os_literal

class os_literal

This class is part of the ObjectStore metaobject protocol, which provides access to ObjectStore schemas. Instances of this class represent literals that designate values. They can be used as actual parameters of class templates.

os_literal::create_char()

static os_literal& create_char(char);

Creates a literal representing the specified **char**.

os_literal::create_enum_literal()

static os_literal& create_enum_literal(os_enumerator_literal&);

Creates a literal representing the specified enumerator.

os_literal::create_pointer_literal()

static os_literal& create_pointer_literal(os_pointer_literal&);

Creates a literal representing the specified pointer.

os_literal::create_signed_char()

static os_literal& create_signed_char(signed char);

Creates a literal representing the specified **signed char**.

os_literal::create_signed_int()

static os_literal& create_signed_int(signed int);

Creates a literal representing the specified **signed int**.

os_literal::create_signed_long()

static os_literal& create_signed_long(signed long);

Creates a literal representing the specified **signed long**.

os_literal::create_signed_short()

static os_literal& create_signed_short(signed short);

Creates a literal representing the specified **signed short**.

os_literal::create_unsigned_char()

static os_literal& create_unsigned_char(unsigned char);

Creates a literal representing the specified **unsigned char**.

os_literal::create_unsigned_int()

static os_literal& create_unsigned_int(unsigned int);

Creates a literal representing the specified **unsigned int**.

os_literal::create_unsigned_long()

static os_literal& create_unsigned_long(unsigned long);

Creates a literal representing the specified **unsigned long**.

os_literal::create_unsigned_short()

static os_literal& create_unsigned_short(unsigned short);

Creates a literal representing the specified **unsigned short**.

os_literal::create_wchar_t()

static os_literal& create_wchar_t(wchar_t);

Creates a literal representing the specified **wchar_t**.

os_literal::get_char_value()

char get_char_value() const;

Returns the value designated by the specified literal.

os_literal::get_enum_literal()

const os_enumerator_literal& get_enum_literal() const;

Returns a reference to the **const os_enumerator_literal** designated by the specified literal.

os_enumerator_literal& get_enum_literal();

Returns a reference to the **os_enumerator_literal** designated by the specified literal.

os_literal::get_kind()

```
enum os_literal_kind {
    Enumerator_literal,
    Function_literal,
    Function_literal_template,
    Unsigned_char_literal,
    Signed_char_literal,
    Unsigned_short_literal,
    Signed_short_literal,
    Integer_literal,
    Unsigned_integer_literal,
    Signed_long_literal,
    Unsigned_long_literal,
    Char_literal,
    Pointer_literal,
    Wchar_t_literal,
};
os_literal_kind get_kind() const;
```

Returns an enumerator indicating the kind of the specified literal.

os_literal::get_pointer_literal()

const os_pointer_literal& get_pointer_literal() const;

Returns a reference to the **const os_pointer_literal** designated by the specified literal.

os_pointer_literal& get_pointer_literal();

Returns a reference to the **os_pointer_literal** designated by the specified literal.

os_literal::get_signed_integral_value()

long get_signed_integral_value() const;

Returns the value designated by the specified literal.

os_literal::get_type()

const os_type& get_type() const;

Returns the type of the value designated by the specified literal.

os_literal::get_unsigned_integral_value()

long get_unsigned_integral_value() const;

Returns the value designated by the specified literal.

os_literal::get_wchar_t_value()

wchar_t get_wchar_t_value() const;

Returns the value designated by the specified literal.

os_literal_template_actual_arg

class os_literal_template_actual_arg : public os_template_actual_arg

This class is part of the ObjectStore metaobject protocol, which provides access to ObjectStore schemas. Instances of this class represent values that are actual parameters of class templates.

os_literal_template_actual_arg::create()

static os_literal_template_actual_arg& create(os_literal*);

Creates an actual parameter consisting of the specified literal.

os_literal_template_actual_arg::get_literal()

const os_literal &get_type() const;

Returns a reference to a **const** literal, the literal of which the actual parameter consists.

os_literal &get_literal();

Returns a reference to a non-**const** literal, the literal of which the actual parameter consists.

os_literal_template_actual_arg::set_literal()

void set_literal(os_literal&);

Sets the literal of which the actual parameter consists.

os_lock_timeout_exception

Instances of this class contain information on the circumstances preventing acquisition of a lock within a specified timeout period. An exception of this type can be signaled by processes that have called the **set_readlock_timeout()** or **set_writelock_timeout()** member of **os_segment**, **os_database**, or **objectstore**. A pointer to an instance of this class can be returned by **objectstore::acquire_ lock()**.

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_lock_timeout_exception::get_application_names()

os_char_p *get_application_names();

Returns an array of strings naming the applications preventing lock acquisition. This array is parallel to the arrays returned by **get_hostnames()** and **get_pids()**; that is, the i^{th} element of **get_ application_names()** contains information about the same process as the i^{th} elements of **get_hostnames()** and **get_pids()**. The member function **number_of_blockers()** returns the number of elements in these arrays. Deleting the **os_lock_timeout_exception** deallocates the arrays.

os_lock_timeout_exception::get_fault_addr()

void *get_fault_addr();

Returns the address on which ObjectStore faulted, causing the database access leading to the attempted lock acquisition.

os_lock_timeout_exception::get_hostnames()

os_char_p *get_hostnames();

Returns an array of strings naming the host machines running the applications preventing lock acquisition. This array is parallel to the arrays returned by **get_application_names()** and **get_pids()**; that is, the i^{th} element of **get_hostnames()** contains information about the same process as the i^{th} elements of **get_application_ names()** and **get_pids()**. The member function **number_of_**

blockers() returns the number of elements in these arrays. Deleting the **os_lock_timeout_exception** deallocates the arrays.

os_lock_timeout_exception::get_lock_type()

os_int32 get_lock_type();

Returns a value (**os_read_lock** or **os_write_lock**) indicating the type of lock ObjectStore was requesting when the timeout occurred.

os_lock_timeout_exception::get_pids()

os_unsigned_int32 *get_pids();

Returns an array of integers indicating the process IDs of the processes preventing lock acquisition. This array is parallel to the arrays returned by **get_application_names()** and **get_hostnames()**; that is, the i^{th} element of **get_pids()** contains information about the same process as the i^{th} elements of **get_application_names()** and **get_hostnames()**. The member function **number_of_blockers()** returns the number of elements in these arrays. Deleting the **os_ lock_timeout_exception** deallocates the arrays.

os_lock_timeout_exception::number_of_blockers()

os_int32 number_of_blockers();

Returns the number of processes preventing lock acquisition.

os_member

This class is part of the ObjectStore metaobject protocol, which provides access to ObjectStore schemas. An instance of this class represents a class member.

Programs using this class must include **<ostore/ostore.hh>**, followed by **<ostore/coll.hh>** (if used), followed by **<ostore/mop.hh>**.

os_member::Access_modifier

This enumerator is a possible return value from **os_ member::kind()**, indicating an access modification to an inherited member. See **os_access_modifier** on page 44.

os_member::defining_class()

const os_class_type &defining_class() const;

Returns a reference to a **const os_class_type**, the class that defines the specified member.

os_class_type &defining_class();

Returns a reference to a non-**const os_class_type**, the class that defines the specified member.

os_member::Field_variable

This enumerator is a possible return value from **os_ member::kind()**, indicating a bit field. See **os_instantiated_class_ type** on page 145.

os_member::Function

This enumerator is a possible return value from **os_ member::kind()**, indicating a member function. See **os_member_ function** on page 159.

os_member::get_access()

int get_access() const;

Returns an enumerator describing the access to the specified member, **os_member::Private**, **os_member::Protected**, or **os_ member::Public**.

os_member::get_defining_class()

const os_class_type &defining_class() const;

Returns a reference to a **const os_class_type**, the type that defines the specified member.

const os_class_type &defining_type() const;

Returns a reference to a non-**const os_class_type**, the type that defines the specified member.

os_member::get_kind()

int get_kind() const;

Returns an enumerator indicating the subtype of **os_member** of which the specified object is a direct instance. The possible return values are **os_member::Variable**, **os_member::Function**, **os_member::Type**, **os_member::Access_modifier**, **os_member::Field_variable**, **os_member::Namespace**, or **os_member::Relationship**.

os_member::is_unspecified()

os_boolean is_unspecified() const;

Returns nonzero (that is, true) if and only if the specified **os_member** is the *unspecified member*. Some **os_member-**valued attributes in the metaobject protocol are required to have values in a consistent schema, but might lack values in the transient schema, before schema installation or evolution is performed. The get function for such an attribute returns a reference to an **os_member**. The fact that a reference rather than pointer is returned indicates that the value is required in a consistent schema. In the transient schema, if such an attribute lacks a value (because you have not yet specified it), the get function returns the unspecified member. This is the only **os_member** for which **is_unspecified()** returns nonzero.

os_member::Namespace

This enumerator is a possible return value from **os_member::get_kind()**, indicating a member function. See **os_member_namespace** on page 164.

os_member::operator const os_access_modifier&()

operator const os_access_modifier&() const;

Provides for safe casts from **const os_member** to **const os_access_modifier&**. If the cast is not permissible, err_mop_illegal_cast is signaled.

os_member::operator const os_field_member_variable&()

operator const os_field_member_variable&() const;

Provides for safe casts from **const os_member** to **const os_field_member_variable&**. If the cast is not permissible, err_mop_illegal_cast is signaled.

os_member::operator const os_member_function&()

operator const os_member_function&() const;

Provides for safe casts from **const os_member** to **const os_member_function&**. If the cast is not permissible, err_mop_illegal_cast is signaled.

os_member::operator const os_member_type&()

operator const os_member_type&() const;

Provides for safe casts from **const os_member** to **const os_member_type&**. If the cast is not permissible, err_mop_illegal_cast is signaled.

os_member::operator const os_member_variable&()

operator const os_member_variable&() const;

Provides for safe casts from **const os_member** to **const os_member_variable&**. If the cast is not permissible, err_mop_illegal_cast is signaled.

os_member::operator const os_relationship_member_variable&()

operator const os_relationship_member_variable&() const;

Provides for safe casts from **const os_member** to **const os_relationship_member_variable&**. If the cast is not permissible, err_mop_illegal_cast is signaled.

os_member::operator os_access_modifier&()

operator os_access_modifier&();

Provides for safe casts from **os_member** to **os_access_modifier&**. If the cast is not permissible, err_mop_illegal_cast is signaled.

os_member::operator os_field_member_variable&()

operator os_field_member_variable&();

Provides for safe casts from **os_member** to **os_field_member_variable&**. If the cast is not permissible, err_mop_illegal_cast is signaled.

os_member::operator os_member_function&()

operator os_member_function&();

Provides for safe casts from **os_member** to **os_member_function&**. If the cast is not permissible, err_mop_illegal_cast is signaled.

os_member::operator os_member_type&()

operator os_member_type&();

Provides for safe casts from **os_member** to **os_member_type&**. If the cast is not permissible, err_mop_illegal_cast is signaled.

os_member::operator os_member_variable&()

operator os_member_variable&();

Provides for safe casts from **os_member** to **os_member_variable&**. If the cast is not permissible, err_mop_illegal_cast is signaled.

os_member::operator os_relationship_member_variable&()

operator os_relationship_member_variable&();

Provides for safe casts from **os_member** to **os_relationship_member_variable&**. If the cast is not permissible, err_mop_illegal_cast is signaled.

os_member::Private

This enumerator is a possible return value from **os_member::get_access()**, indicating **private** access.

os_member::Protected

This enumerator is a possible return value from **os_member::get_access()**, indicating **protected** access.

os_member::Public

This enumerator is a possible return value from **os_member::get_access()**, indicating **public** access.

os_member::Relationship

This enumerator is a possible return value from **os_member::kind()**, indicating a relationship (inverse) member. See **os_relationship_member_variable** on page 267.

os_member::set_access()

void set_access(int);

Specifies an enumerator describing the access to the specified member, **os_member::Private**, **os_member::Protected**, or **os_member::Public**.

os_member::Type

This enumerator is a possible return value from **os_member::kind()**, indicating that the specified member is a nested type definition. See **os_member_type** on page 165.

os_member::Variable

This enumerator is a possible return value from **os_member::kind()**, indicating a data member. See **os_member_variable** on page 166.

os_member_function

class os_member_function : public os_member

This class is part of the ObjectStore metaobject protocol, which provides access to ObjectStore schemas. Instances of this class represent member functions. **os_member_function** is derived from **os_member**.

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.

Programs using this class must include **<ostore/ostore.hh>**, followed by **<ostore/coll.hh>** (if used), followed by **<ostore/mop.hh>**.

os_member_function::create()

```
static os_member_function& create(
    const char* name,
    os_function_type*
);
```

Creates a member function with the specified name and of the specified type.

os_member_function::get_call_linkage()

```
os_call_linkage get_call_linkage() const;
```

Returns **os_member_function::No_linkage**, **os_member_function::C_linkage**, **os_member_function::C_plus_plus_linkage**, or **os_member_function::Fortran_linkage**.

os_member_function::get_function_kind()

```
enum os_function_kind {
    Regular,
    /* applicable only if it is a member function */
    Constructor, Destructor,
    Cast_op, /* the return type gives the cast type */
    /* the operators that can be overloaded */
    New_op, Delete_op,
    Plus_op, Minus_op, Mul_op, Div_op, Mod_op,
    Xor_op, And_op, Or_op, Comp_op,
    Not_op, Assign_op, Lt_op, Gt_op,
    Plus_assign_op, Minus_assign_op, Mul_assign_op,
    Div_assign_op, Mod_assign_op,
    Xor_assign_op, And_assign_op, Or_assign_op,
    Lsh_op, Rsh_op,
    Lsh_assign_op, Rsh_assign_op,
    Eq_op, Neq_op, Le_op, Ge_op,
    And_and_op, Or_or_op,
    Inc_op, Dec_op, Comma_op,
    Member_deref_op, Deref_op,
    Paren_op, Subscript_op,
    Vec_new_op, Vec_delete_op
};
```

```
os_function_kind get_function_kind() const;
```

Returns an enumerator indicating what kind of function the specified member function is.

os_member_function::get_name()

```
const char *get_name() const;
```

Returns the name of the specified member.

os_member_function::get_source_position()

```
void get_source_position(
    const char* &file,
    os_unsigned_int32 &line
) const;
```

Returns the source position associated with the specified function.

os_member_function::get_type()

```
const os_function_type &get_type() const;
```

Returns an **os_function_type&**, which contains information about the function, including its return type and argument list.

os_member_function::is_const()

```
os_boolean is_const() const;
```

Returns nonzero if and only if the specified member function is **const**.

os_member_function::is_inline()

```
os_boolean is_inline() const;
```

Returns nonzero if and only if the specified member function is inline.

os_member_function::is_overloaded()

os_boolean is_overloaded() const;

Returns nonzero if and only if the specified member function is overloaded.

os_member_function::is_pure_virtual()

os_boolean is_pure_virtual() const;

Returns nonzero if and only if the specified member function is pure virtual.

os_member_function::is_static()

os_boolean is_static() const;

Returns nonzero if and only if the specified member function is static.

os_member_function::is_virtual()

os_boolean is_virtual() const;

Returns nonzero if and only if the specified member function is virtual.

os_member_function::is_volatile()

os_boolean is_volatile() const;

Returns nonzero if and only if the specified member function is volatile.

os_member_function::set_call_linkage()

void set_call_linkage(os_call_linkage);

Pass **os_member_function::No_linkage**, **os_member_function::C_linkage**, **os_member_function::C_plus_plus_linkage**, or **os_member_function::Fortran_linkage**.

os_member_function::set_is_const()

void set_is_const(os_boolean);

1 specifies that the member function is **const**; **0** specifies that it is non-**const**.

os_member_function::set_is_inline()

void set_is_inline(os_boolean);

1 specifies that the member function is inline; **0** specifies that it is not inline.

os_member_function::set_is_overloaded()

void set_is_overloaded(os_boolean);

1 specifies that the member function is overloaded; **0** specifies that it is not.

os_member_function::set_is_pure_virtual()

void set_is_pure_virtual(os_boolean);

1 specifies that the member function is a pure virtual function; **0** specifies that it is not.

os_member_function::set_is_static()

void set_is_static(os_boolean);

1 specifies that the member function is a static function; **0** specifies that it is not.

os_member_function::set_is_virtual()

void set_is_virtual(os_boolean);

1 specifies that the member function is a virtual function; **0** specifies that it is not.

os_member_function::set_is_volatile()

void set_is_volatile(os_boolean);

1 specifies that the member function is volatile; **0** specifies that it is not.

os_member_function::set_name()

void set_name(const char* name);

Sets the name of the specified member.

os_member_function::set_source_position()

```
void set_source_position(
    const char* file,
```

os_unsigned_int32 line
);

Sets the source position associated with the specified function.

os_member_function::set_type()

void set_type(os_function_type&);

Sets the return type of the specified member function.

os_member_namespace

class os_member_namespace : public os_member

This class is part of the ObjectStore metaobject protocol. Because namespaces can be enclosed in namespaces they must occur as members of namespaces. The class **os_member_namespace** is used to represent namespaces as members.

os_member_namespace::create()

static os_member_namespace& create(os_namespace*) ;

os_member_namespace::get_namespace()

const os_namespace& get_namespace() const ;

os_member_namespace::set_namespace

os_namespace& get_namespace() ;

void set_namespace(os_namespace&);

os_member_type

class os_member_type : public os_member

This class is part of the ObjectStore metaobject protocol, which provides access to ObjectStore schemas. An instance of this class represents a member type definition, that is, a type definition that is nested within a class. **os_member_type** is derived from **os_member**.

os_member_type::create()

static os_member_type& create(os_type*);

Creates a member typedef for the specified **os_type**.

os_member_type::get_type()

const os_type &get_type() const;

Returns the type defined by the specified member typedef.

os_member_type::set_type()

void set_type(os_type&);

Sets the type defined by the specified member typedef.

os_member_variable

class os_member_variable : public os_member

This class is part of the ObjectStore metaobject protocol, which provides access to ObjectStore schemas. Instances of this class represent data members. **os_member_variable** is derived from **os_member**.

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.

Programs using this class must include **<ostore/ostore.hh>**, followed by **<ostore/coll.hh>** (if used), followed by **<ostore/mop.hh>**.

os_member_variable::create()

```
static os_member_variable &create(
    const char *name,
    os_type *value_type
);
```

Creates an **os_member_variable**. The arguments specify the initial values for the attributes **name** and **value_type**. The initial values for the remaining attributes are as follows:

Attribute	*Value*
storage_class	**os_member_variable::Regular**
is_field	**0**
is_static	**0**
is_persistent	**0**

os_member_variable::get_name()

```
const char *get_name() const;
```

Returns the name of the specified member.

os_member_variable::get_type()

```
const os_type &get_type() const;
```

Returns a reference to a **const os_type**, the value type of the specified member.

os_type &get_type();

Returns a reference to a non-**const os_type**, the value type of the specified member.

os_member_variable::get_size()

os_unsigned_int32 get_size() const;

Returns the size in bytes occupied by the specified member. Signals err_mop if the specified member is an **os_field_member_variable**.

os_member_variable::get_offset()

os_unsigned_int32 get_offset() const;

Returns the offset in bytes of the specified member within its defining class. Signals err_mop if the specified member is an **os_field_member_variable**, or is a static or persistent member.

os_member_variable::get_source_position()

```
void get_source_position(
    const char* &file,
    os_unsigned_int32 &line
) const;
```

Returns the source position associated with the specified member.

os_member_variable::get_storage_class()

enum os_storage_class { Regular, Persistent, Static } ;

os_storage_class get_storage_class() const;

Returns an enumerator indicating the storage class of the specified member: **os_member_variable::Regular**, **os_member_variable::Persistent**, or **os_member_variable::Static**.

os_member_variable::is_field()

os_boolean is_field() const;

Returns **1** if and only if the specified member is an **os_field_member_variable**.

os_member_variable::is_static()

os_boolean is_static() const;

Returns **1** if and only if the specified member is a static data member.

os_member_variable::is_persistent()

os_boolean is_persistent() const;

Returns **1** if and only if the specified member is a persistent data member.

os_member_variable::operator const os_field_member_variable&()

operator const os_field_member_variable&() const;

Provides for safe casts from **const os_member_variable** to **const os_field_member_variable&**. If the cast is not permissible, err_mop_illegal_cast is signaled.

os_member_variable::operator const os_relationship_member_variable&()

operator const os_relationship_member_variable&() const;

Provides for safe casts from **const os_member_variable** to **const os_relationship_member_variable&**. If the cast is not permissible, err_mop_illegal_cast is signaled.

os_member_variable::operator os_field_member_variable&()

operator os_field_member_variable&();

Provides for safe casts from **os_member_variable** to **os_field_member_variable&**. If the cast is not permissible, err_mop_illegal_cast is signaled.

os_member_variable::operator os_relationship_member_variable&()

operator os_relationship_member_variable&();

Provides for safe casts from **os_member_variable** to **os_relationship_member_variable&**. If the cast is not permissible, err_mop_illegal_cast is signaled.

os_member_variable::set_name()

void set_name(const char *);

Specifies the name of the specified member. ObjectStore copies the character array pointed to by the argument.

os_member_variable::set_offset()

void set_offset(os_unsigned_int32);

Sets the offset in bytes of the specified member within its defining class. Signals err_mop if the specified member is an **os_field_ member_variable**, or is a static or persistent member.

os_member_variable::set_source_position()

```
void set_source_position(
    const char* file,
    os_unsigned_int32 line
);
```

Sets the source position associated with the specified member.

os_member_variable::set_storage_class()

void set_storage_class(os_unsigned_int32);

Specifies an enumerator indicating the storage class of the specified member: **os_member_variable::Regular**, **os_member_ variable::Persistent**, or **os_member_variable::Static**.

os_member_variable::set_type()

set_type(os_type &);

Specifies the value type of the specified member.

os_mop

This class is part of the ObjectStore metaobject protocol, which provides access to ObjectStore schemas. The members provided concern the transient schema.

Programs using this class must include **<ostore/ostore.hh>**, followed by **<ostore/coll.hh>** (if used), followed by **<ostore/mop.hh>**.

os_mop::bind()

```
void os_mop::bind (
    const char* hetero_set,
    os_schema_options* schema_options,
    os_boolean make_neutral_changes,
    os_boolean allow_schema_reorg,
    const char** neutral_output
);
```

Causes all classes in the transient schema to be bound for the current architecture. A default invocation of this binding function occurs automatically when classes are installed into a database schema. This interface allows the binding to occur independently, and allows additional functionality beyond the default behavior to be invoked.

It is important to consider the effects of heterogeneity on schema neutralization. See ossg Neutralization Options in Chapter 5 of *ObjectStore Building C++ Interface Applications* for detailed information.

The **hetero_set** argument can specify any heterogeneity set supported for the current platform, or can be set to null if no heterogenity is requested.

The **schema_options** argument specifies the compiler options and pragmas to be used on this and other platforms.

The **make_neutral_changes** argument controls whether **os_mop** automatically modifies the schema to make it neutral. If **make_neutral_changes** is set to false and the schema is not neutral, the exception err_mop_not_neutral is signaled.

The **allow_schema_reorg** argument permits **os_mop** to make more complex modifications in order to ensure schema neutralization.

The **neutral_output** argument allows the caller to receive a string containing a description of the neutralization changes and / or failures encountered. The caller must delete the returned string.

Neutralization failures

If the schema is not neutral and cannot be made neutral for some reason, the exception err_mop_cannot_neutralize is signaled. This could occur if

- A schema construct is used that is incompatible with a selected heterogeneity set (such as using virtual base classes with the **set1** heterogeneity set).
- You fail to specify **allow_schema_reorg** when necessary (virtual base classes often require this).
- You fail to specify a **schema_options** argument with the necessary options.

See *ObjectStore Building C++ Interface Applications*, Chapter 5, Building Applications for Use on Multiple Platforms, for more details on schema neutralization options and regulations.

os_mop::copy_classes()

```
static void copy_classes (
    const os_schema &schema,
    os_const_classes &classes
);
```

Copies the specified classes into the transient schema. If any of the given classes is not well formed or is not from the given schema, or the given schema is the transient schema, an exception is raised.

os_mop::current()

```
static os_schema &current ();
```

Returns the schema currently bound. The bound schema is the schema in which dynamically created types are deposited. After initialization of schema services, the current schema is bound to the schema found in the transient database.

os_mop::find_namespace()

```
static os_namespace *find_namespace (const char* name);
```

Returns the **os_namespace** associated with the given name in the **os_schema** denoted by **os_mop::current()**.

os_mop::find_type()

static os_type *find_type(const char *name);

Returns a pointer to the type in the transient schema with the specified name; returns **0** if there is no such type.

os_mop::get_transient_schema()

static os_schema &get_transient_schema();

Returns a reference to the transient schema.

os_mop::get_failure_classes()

os_classes osmop::get_failure_classes ();

Following a call to **bind()** and before a call to **os_mop::reset** or **os_ database_schema::install()**, the function **get_failure_classes()** returns the classes for which no valid neutralization was found.

This list should be empty except after a call to **bind()** that results in err_mop_cannot_neutralize's being signaled. Note that the neutralization failure of a class can hide further neutralization failures because no attempt is made to neutralize types derived from or that embed failing classes.

os_mop::get_neutralized_classes()

os_classes osmop::get_neutralized_classes();

Following a call to **bind()** and before a call to **os_mop::reset** or **os_ database_schema::install()**, the function **get_neutralized_classes()** returns the classes for which changes were required. If the previous call to bind resulted in err_mop_cannot_neutralize's being signaled, this list is not necessarily complete.

os_mop::initialize()

static void initialize();

Must be called before you use the transient schema, that is, before you create any schema objects and before you copy any classes into the transient schema.

os_mop::initialize_object_metadata()

```
static void os_mop::initialize_object_metadata(
    void *object,
    const char *type_name);
```

This interface initializes the compiler metadata, if any, associated with the object. The object instance can be transient or persistent; however, the schema for its class must be present in the application schema. If the object is transient, the **type_name** argument must be nonnull, and indicates the name of the class of the object.

If the object is persistent, the **type_name** can be null. If nonnull, it must match exactly the real type of the object; otherwise an exception is generated. For example, names returned by the **os_types** or **os_mop** subsystem are safe to use, and will match correctly. The best method is to pass a null **type_name** when the instance is persistent.

This call must be made while inside a transaction. The object pointer must be a pointer to a valid top-level object. Pointers to embedded objects will generate an exception in the case of persistent pointers where this case can be verified. In the case of transient instances, no such checking is possible, and a bad initialization could result.

Top-level arrays must be initialized one element at a time. The object pointer must point beyond any vector headers in a top-level array. Embedded arrays within objects are initialized correctly. Also, any compiler metadata inside a union with discriminants will *not* be initialized. It is very difficult to arrange to call union discriminant functions during this initialization.

Virtual base class pointers, vector headers, and any other compiler metadata are not affected by this interface. It also has no effect on normal class data.

The equivalent C interface is found in Chapter 6, C Library Interface, on page 393.

os_mop::reset()

static void reset ();

Reset the portion of schema services responsible for the access and construction of schema types through the MOP interface. After this operation, the current schema is empty

os_named_indirect_type

class os_named_indirect_type : public os_anonymous_indirect_type

This class is part of the ObjectStore metaobject protocol, which provides access to ObjectStore schemas. An instance of this class represents a C++ typedef. This class is derived from **os_anonymous_indirect_type**. Performing **os_anonymous_indirect_type::get_target_type()** on an **os_named_indirect_type** results in the type named by the typedef.

os_named_indirect_type::create()

```
static os_named_indirect_type& create(
    os_type* target,
    const char*
);
```

Creates an **os_named_indirect_type**.

os_named_indirect_type::get_name()

```
const char *get_name() const;
```

Returns the name of the specified typedef.

os_named_indirect_type::get_source_position()

```
void get_source_position(
    const char* &file,
    os_unsigned_int32 &line
) const;
```

Returns the source position associated with the specified typedef.

os_named_indirect_type::set_name()

```
const char *set_name() const;
```

Sets the name of the specified typedef.

os_named_indirect_type::set_source_position()

```
void set_source_position(
    const char* file,
    os_unsigned_int32 line
);
```

Sets the source position associated with the specified typedef.

os_namespace

This class is part of the ObjectStore metaobject protocol.

os_namespace::create()

static os_namespace& create(const char* name);

Creates a namespace typedef for the specified name.

static os_namespace& create(const char* name, os_members&) ;

Creates a namespace typedef for the specified name and members.

os_namespace::get_enclosing_namespace()

const os_namespace* get_enclosing_namespace () const ;

os_namespace* get_enclosing_namespace ();

Returns the enclosing namespace if one exists, otherwise returns **0**.

os_namespace::get_members()

os_members get_members () ;

os_const_members get_members () const ;

os_namespace::get_name()

const char* get_name () const ;

os_namespace::set_members()

void set_members(os_members&) ;

os_namespace::set_name()

void set_name(const char*) ;

os_namespace::set_enclosing_namespace()

const os_namespace* set_enclosing_namespace () const ;

os_namespace* set_enclosing_namespace ();

Returns the enclosing namespace if one exists, otherwise returns **0**.

os_notification

Objects of class **os_notification** represent notifications for sending and receiving. A notification object embodies a database location (**os_reference**), a signed 32-bit integer **kind**, and a null-terminated C string.

os_notification::os_notification()

```
os_notification(
    const os_reference& ref,
    os_int32 kind=0,
    const char *string=0
);
```

Notifications can be created using a constructor that specifies all these elements. The **os_notification** object copies its string argument when the object is created, and deletes the storage for the string copy when it is deleted.

os_notification();

Notifications can also be allocated in arrays (see **os_subscription** on page 318**)**. The default constructor for **os_notification** produces an uninitialized notification.

os_notification::assign()

```
void assign(
    const os_reference& ref,
    os_int32 kind=0,
    const char *string=0
);
```

Notifications can be reassigned using the assign member function. This is most useful when allocating arrays of notifications:

When passing database locations to **os_notification** member functions, you do not need to explicitly convert to **os_reference**. You can pass pointers or **os_Reference<X>**. These are converted by C++ automatically.

You can retrieve the components of a notification using the following accessor functions.

os_notification::get_database()

os_database *get_database()const;

Retrieves the database associated with the notification.

os_notification::get_reference()

os_reference get_reference()const;

Retrieves the reference associated with the notification.

os_notification::get_kind()

os_int32 get_kind()const;

Retrieves the kind associated with the notification.

os_notification::get_string()

const char *get_string()const;

Retrieves the string associated with the notification.

notify_immediate and **notify_on_commit** member functions provide shortcuts for the static member functions also defined on this class:

void notify_immediate();
void notify_on_commit();

A public enumeration in class **os_notification** represents the maximum string length usable in notifications.

enum {maximum_string_length = 16383};

A public enumeration in class **os_notification** represents the maximum notification queue size.

enum {maximum_notification_queue_length = 512};

os_notification::_get_fd();

Returns a file descriptor that can be used to detect whether any notifications exist. The only legal operation on this file descriptor is to call **select()** or **poll()**, to determine if any data has been received. If data has been received, then a notification has been queued for this application. It can be retrieved using **os_ notification::receive()**.

After retrieving a notification, you can test for further notifications again using **select()** or **poll()**. (That is, **os_ notification::receive()** resets the **notification_fd** to the not ready state unless there are further notifications pending.

This function is not available on all platforms and configurations. This is because file descriptors are not used portably on all platforms. If notifications are not delivered using an **fd** mechanism, this function returns **–1**.

Using this **fd** for any purpose other than **poll()** or **select()** could cause unexpected application behavior. The Cache Manager and/or Server could disconnect from the client.

os_notification::notify_immediate()

```
static void os_notification::notify_immediate(
    const os_reference &ref,
    os_int32 kind = 0,
    const char *string = 0,
    os_int32 *n_queued = 0
);
```

```
static void os_notification::notify_immediate(
    const os_notification *notifications,
    os_int32 n_notifications = 1,
    os_int32 *n_queued = 0
);
```

Posts a notification to the object specified by **ref**. The database associated with **ref** must be open. The **kind** and **string** arguments are arguments sent to the receiving process.

Note: The **kind** argument must be greater than or equal to zero. Negative kinds are reserved for future use by ObjectStore.

If the string specified is null (0), it is received as an empty string ("").

If supplied, the **n_queued** argument is a signed 32-bit integer (or array of **n_notification** integers). This integer is set to the number of receiving processes to which notifications were queued. Note that because notifications are asynchronous, no guarantees can be made that the process will ever receive the notification. (The receiving process might terminate before receiving the notification, it might never check for notifications, the Server might crash, the Cache Manager might crash, or the notification queue could overflow.)

In the second (array) form of **notify_immediate**, the **n_queued** argument, if specified, is an **os_int32** array at least **n_notifications** long. The elements of the array are set to the number of receiving processes for each notification specified.

Each call results in a single RPC call to each ObjectStore Server. It is significantly more efficient to make one call with an array of notifications than to make many calls with each notification.

If the caller does not require the **n_queued** information, it should leave **n_queued** defaulted to 0. This could result in better performance in future releases.

The **notify_immediate** operation is not atomic. That is, if an error is signaled, the status of notifications is undefined. For example, notifications might have been successfully delivered to one Server before a second Server signals an error with respect to its notifications.

os_notification::notify_on_commit()

```
static void os_notification::notify_on_commit(
    const os_notification *notifications,
    os_int32 n_notifications = 1);
```

Queues a commit-time notification to the object specified by **ref**. The database associated with **ref** must be open, and there must be a transaction in progress. The notification is delivered when, if, and only if,

- No enclosing transaction aborts.
- The enclosing top-level transaction commits. The **kind** and **string** arguments are sent to the receiving processes after the commit completes.

If the string specified is null (**0**), it is received as an empty string ("").

Notification delivery and commit are an atomic operation from the perspective of the process sending the notification. That is, if the commit succeeds, the notifications are guaranteed to be sent, even if the sending application crashes. Note however that the notifications themselves are transient, and might be lost if there is a Server failure, Cache Manager failure, notification queue overflow, or if a receiving process dies.

The notifications are matched with subscriptions immediately after the commit succeeds. Because of this, there is no way to determine how many processes have been queued to receive notifications.

If a deadlock occurs during a stack transaction, ObjectStore aborts and automatically restarts the transaction. In this case, because the transaction aborted, commit-time notifications are discarded, and execution resumes at the beginning of the stack transaction.

os_notification::notify_on_commit does not immediately perform an RPC call to the Server. If there are any calls to **os_ notification::notify_on_commit** during a top-level transaction, and the transaction commits, there is one additional RPC call to each Server at commit time.

In some read-only transactions, the ObjectStore client does not normally have to communicate with Servers. If **os_ notification::notify_on_commit** is called during such a read-only transaction, the Server must be contacted during the commit.

If a database is closed after a **notify_on_commit** but before committing the transaction, the notification will still be delivered on successful commit. The database must be open when **notify_ on_commit** is called.

os_notification::queue_status()

```
static void os_notification::queue_status(
    os_unsigned_int32 &queue_size,
    os_unsigned_int32 &count_pending_notifications,
    os_unsigned_int32 &count_queue_overflows
);
```

Returns information on the notification queue for the current process. If **count_pending_notifications** is greater than zero, notifications are pending. This function can be used as a polling function to see if there are notifications without actually retrieving them. It does not lock out other ObjectStore operations in other threads.

Generally, applications should call this at least once after each notification is retrieved, to ensure that there are no queue overflows, and perform appropriate actions if they do occur.

Values returned are as follows:

queue_size	The size of the notification queue, as set by **os_notification::set_notification_queue_size()**, **OS_NOTIFICATION_QUEUE_SIZE**, or defaulted.
count_pending_notifications	The number of notifications currently in the queue. That is, notifications that have not yet been received by the process using **os_notification::receive()**.
count_queue_overflows	The number of notifications discarded since the process started. A value of 0 indicates that no notifications have ever been discarded since this process began.

os_notification::receive()

```
static os_boolean os_notification::receive(
    os_notification *&notification,
    os_int32 timeout = -1
);
```

Gets the next notification from the notification queue, if available. If a notification is available, it returns true, and places the notification in the first argument. Otherwise, it returns false, and the first argument is unmodified.

If the notification queue is currently empty, the function waits as specified by the timeout argument. A value of **–1** indicates to wait forever until a notification is received. A value of **0** indicates to return false immediately. A positive integer indicates to wait the specified number of milliseconds. On some platforms, this value is rounded up to the next higher number of seconds.

The notification returned is allocated dynamically in transient storage. When the application finishes using it, it can be deleted using the C++ **delete** operator. (This causes the notification string to be deleted as well.)

os_notification::receive() uses operating system primitives for waiting; it does not spin, polling for notifications. Users normally call this function in a separate thread that exists specifically to receive notifications.

Only one thread can call **os_notification::receive()** at any one time. It is an error to call it in multiple threads simultaneously.

Only **os_notification::receive()** and **os_notification::queue_status()** can be called asynchronously with other ObjectStore operations. All other APIs, including the **os_notification** accessors, are subject to normal thread-locking rules. This means that the retrieved notifications cannot be accessed in concurrent threads without locking out ObjectStore threads.

If **os_notification::receive()** is called before subscribing to notifications, it returns false immediately, regardless of the timeout argument. This is to avoid deadlocks in some situations involving multiple threads. To avoid this, ensure that **os_notification::subscribe()** or **os_notification::_get_fd()** is called before calling **os_notification::receive()**, or before launching a thread that calls **os_notification::receive()**.

With long strings, **os_notification::receive()** might have to wait slightly for the entire string, even if **timeout==0** is specified.

os_notification::set_queue_size()

static void os_notification::set_queue_size(os_int32u queue_size);

Sets the size of the notification queue for a process. It must be called prior to **os_notification::subscribe()** or **os_notification::_get_fd()**. If this function is not called, the queue size is determined by the value of the **OS_NOTIFICATION_QUEUE_SIZE** environment variable. If the environment variable is not set, the queue size is set to a default value, currently 50.

A public enumeration in class **os_notification** represents the maximum notification queue size.

enum {maximum_notification_queue_length = 512};

Notification queues are part of the ObjectStore Cache Manager process.

os_notification::subscribe()

```
static void os_notification::subscribe(const os_subscription *sub, os_int32 count = 1);
static void os_notification::subscribe(const os_database *db);
static void os_notification::subscribe(const os_segment *seg);
static void os_notification::subscribe(const os_object_cluster *clus);
static void os_notification::subscribe(const os_reference &ref, os_int32 n_bytes = 1);
```

These functions all subscribe to notifications. You can subscribe to any notification in a database, segment, cluster, event range, or reference. A subscription in a database, segment, or cluster applies to all addresses in the database, segment, or cluster, even addresses that have not yet been allocated.

The first function lets you subscribe to one **os_subscription**, or an array of **os_subscription**. The **count** argument is the length of the array.

The database must be open. Closing the database immediately unsubscribes all locations associated with the database.

If a database location is subscribed more than once, the notification system behaves as if there were only one subscription on the location. That is, multiple subscriptions on a database location are ignored.

Each call results in a single RPC call to each ObjectStore Server. It is significantly more efficient to make one call with an array of subscriptions than to make many calls with each subscription.

The subscription operation is not atomic. That is, if an error is signaled, the status of subscriptions is undefined. For example, subscriptions might have succeeded on one Server before a second Server signals an error with respect to its subscriptions.

os_notification::unsubscribe()

static void (const os_subscription *sub, os_int32 count = 1);
static void os_notification::unsubscribe(const os_database *db);
static void os_notification::unsubscribe(const os_segment *seg);
static void os_notification::unsubscribe(const os_object_cluster *clus);
static void os_notification::unsubscribe(const os_reference &ref, os_int32 n_bytes = 1);

These functions all unsubscribe database locations for notifications.

If a subscription was made on an entire database, the only way to remove it is to unsubscribe the entire database; you cannot selectively unsubscribe segments or database locations.

If a subscription was made on an entire segment, the only way to remove it is to unsubscribe the entire database, or unsubscribe the entire segment. You cannot selectively unsubscribe database locations within the segment.

If a subscription was made on a cluster or range, ranges can be selectively unsubscribed within the original cluster or range. Unsubscribing an unsubscribed database location has no effect.

Note that closing a database automatically unsubscribes all notifications for the database. Because notifications are processed asynchronously, an application might continue to receive notifications after having unsubscribed.

Each call results in a single RPC call to each ObjectStore Server. It is much more efficient to make one call with an array of subscriptions than to make many calls with each subscription.

The unsubscription operation is not atomic. That is, if an error is signaled, the status of unsubscriptions is undefined. For example, unsubscriptions might have succeeded on one Server before a second Server signals an error with respect to its unsubscription.

Additional notes

- If **os_notification::queue_status()** is called before **os_notification::subscribe()** or **os_notification::_get_fd()**, all values returned are zero.
- **n_queued** can sometimes be larger than queue size.
- In general, **queue_status()** should be called in the same thread as **os_notification::receive().**
- If **queue_status()** is called in another thread while **os_notification::receive()** is in process,
  - **os_notification::receive()** might or might not actually retrieve a notification.
  - **n_queued** might or might not actually reflect the receipt of this notification.
- If **n_queued** is nonzero, **os_notification::receive()** might still sometimes return false, particularly if **receive** is called with a zero timeout. This happens if the Cache Manager cannot empty its queue as fast as the receiving process is calling **os_notification::receive()**.

Network service

When an ObjectStore application uses notifications, it automatically establishes a second network connection to the Cache Manager daemon on the local host. The application uses this connection to receive (and acknowledge the receipt of) incoming notifications from the Cache Manager. (Outgoing notifications are sent to the Server, not the Cache Manager.) See

Modifying Network Port Settings of *ObjectStore Management* for specific details.

Notification errors

The notification APIs do not do complete validation of the arguments passed to them. Malformed arguments can therefore cause segmentation violations or other undefined behavior. See General ObjectStore Exceptions on page 571 for details.

Utilities

ossvrstat currently prints statistics on the number of notifications received and sent by the Server.

oscmstat prints information on notifications queued for clients. This is useful in debugging applications that use notifications.

Detailed information on these user interfaces appears in *ObjectStore Management*.

os_object_cluster

An object cluster is a portion of a segment (see the class **os_segment** on page 295) into which related objects can be clustered at allocation time. An object cluster can contain as little as a single small object or as much as 64 Kbytes of memory.

You can improve application performance by clustering together objects that are expected to be used together by applications. This reduces the number of disk and network transfers the applications will require. Moreover, allocating objects in *different* clusters can increase concurrency, since one process’s lock on an object in one cluster never blocks access by other processes to objects in other clusters (assuming **os_segment::lock_whole_segment** has value **objectstore::lock_as_used** — the default — for the segments containing the clusters).

A cluster of a specified size is created with the **create_object_cluster()** member of the class **os_segment**. A new object can be allocated in a cluster using one of the special overloadings of **::operator new()** (see the description of **::operator new()** on page 367), or, in some cases, a class-specific **new** or **create()** operation tailored for object clustering.

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.

All ObjectStore programs must include the header file **<ostore.hh>**.

os_object_cluster::destroy()

```
void destroy(
    forced_destroy_enum option =
        os_object_cluster::destroy_if_empty
);
```

Destroys the specified **os_object_cluster** if either the cluster contains no nondeleted objects or the **os_int32** argument is the enumerator **os_object_cluster::destroy_always**. If invoked on a nonempty cluster with **os_object_cluster::destroy_if_empty** as argument, the exception err_destroy_cluster_not_empty is signaled.

os_object_cluster::destroy_always

Enumerator, used as an argument to **os_object_cluster::destroy()**, indicating that destruction of the cluster should proceed even if the specified cluster contains nondeleted objects. Destroying a cluster results in deletion of all the objects it contains, but does not result in execution of the destructors for the deleted objects.

os_object_cluster::destroy_if_empty

Enumerator, used as an argument to **os_object_cluster::destroy()**, indicating that destruction of the cluster should proceed only if the specified cluster contains no nondeleted objects. Destroying a cluster results in deletion of all the objects it contains, but does not result in execution of the destructors for the deleted objects.

os_object_cluster::get_info()

```
void get_info(
    os_int32 &cluster_size,
    os_int32 &free,
    os_int32 &contig_free
) const;
```

Modifies **cluster_size** to refer to the size in bytes of the specified **os_object_cluster**; modifies **free** to refer to the number of unallocated bytes in the cluster; modifies **contig_free** to refer to the number of bytes in the largest contiguous portion of unallocated memory in the cluster.

os_object_cluster::is_empty()

```
os_boolean is_empty() const;
```

Returns nonzero (true) if the specified **os_object_cluster** contains no nondeleted objects; returns **0** (false) otherwise.

os_object_cluster::of()

```
static os_object_cluster *of(const void *obj);
```

Returns a pointer to the **os_object_cluster** containing the object pointed to by **obj**. If the object is not contained in a cluster, **0** is returned.

os_object_cluster::segment_of()

```
os_segment *segment_of() const;
```

Returns a pointer to the segment containing the specified **os_ object_cluster**.

os_object_cursor

An object cursor allows retrieval of the objects stored in a specified segment, one object at a time, in an arbitrary order. This order is stable across traversals of the segment, as long as no objects are created or deleted from the segment, and no reorganization is performed (using schema evolution or compaction). Operations are provided for creating a cursor, advancing a cursor, and for testing whether a cursor is currently positioned at an object (or has run off the end of the segment). It is also possible to position a cursor at a specified object in the segment.

In addition, an operation is provided for retrieving the object at which a cursor is positioned, together with an **os_type** representing the type of the object, and, for an object that is an array, a number indicating how many elements it has.

os_object_cursor::current()

```
os_boolean current(
    void* &pointer,
    const os_type* &type,
    os_int32 &count
) const;
```

If the cursor is positioned at an object, returns nonzero (true), sets **pointer** to refer to the address of the object, and sets **type** to refer to an **os_type** representing the object's type. If the object is an array, **count** is set to refer to the number of elements it has; if the object is not an array, **count** is set to **0**. If the cursor is not positioned at an object, **0** (false) is returned, and all three arguments are set to refer to **0**.

os_object_cursor::first()

```
void first();
```

Positions the cursor at the first object in the cursor's associated segment. The object is first in an arbitrary order that is stable across traversals of the segment, as long as no objects are created or deleted from the segment, and no reorganization is performed (using schema evolution or compaction). If there are no objects in the cursor's associated segment, the cursor is positioned at no object.

os_object_cursor::more()

os_boolean more() const;

Returns nonzero (true) if the cursor is positioned at an object. Returns **0** (false) otherwise.

os_object_cursor::next()

void next();

Positions the cursor at the next object in the cursor's associated segment. The object is next in an arbitrary order that is stable across traversals of the segment, as long as no objects are created or deleted from the segment, and no reorganization is performed (using schema evolution or compaction). If the cursor is positioned at no object, err_cursor_at_end is signaled. Otherwise, if there is no next object, the cursor is positioned at no object.

os_object_cursor::os_object_cursor()

os_object_cursor(os_segment *seg);

Creates a new **os_object_cursor** associated with the specified segment. If the segment is empty, the cursor is positioned at no object; otherwise it is positioned at the first object in the cursor's associated segment. The object is first in an arbitrary order that is stable across traversals of the segment, as long as no objects are created or deleted from the segment, and no reorganization is performed (using schema evolution or compaction).

os_object_cursor::set()

void set(const void *ptr);

Positions the cursor at the object containing the address **ptr**. If **ptr** is not an address in the specified cursor's associated segment, signals err_cursor_not_at_object. If **ptr** is in the cursor's associated segment but within unallocated space, the cursor is positioned at no object or is arbitrarily positioned at an object in the segment.

os_object_cursor::~os_object_cursor()

~os_object_cursor();

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

os_pathname_and_segment_number

This class is used by the compactor API to identify segments — see **objectstore::compact()** on page 15. It has two public data members and a constructor. Programs using this function must include **<ostore/compact.hh>**.

os_pathname_and_segment_number::database_pathname

```
const char *database_pathname;
```

The value of this member is the pathname of the database containing the segment identified by the specified **os_pathname_and_segment_number**.

os_pathname_and_segment_number::segment_number

```
os_unsigned_int32 segment_number;
```

The value of this member is the segment number of the segment identified by the specified **os_pathname_and_segment_number**. The segment number of a specified segment can be obtained with **os_segment::get_number()**.

os_pathname_and_segment_number::os_pathname_and_segment_number()

```
os_pathname_and_segment_number(
    const char *db,
    os_unsigned_int32 seg_number
);
```

The constructor takes two arguments: the **db** argument initializes the member **database_pathname**, and the **seg_number** argument initializes the member **segment_number**.

os_pointer_literal

This class is part of the ObjectStore metaobject protocol, which provides access to ObjectStore schemas. An instance of this class represents a C++ pointer literal.

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_pointer_literal::create()

static os_pointer_literal& create(const char*, os_pointer_type*);

Creates an **os_pointer_literal** of the specified name and type.

os_pointer_literal::get_name()

const char *get_name() const;

Returns the name of the specified literal.

os_pointer_literal::get_type()

os_pointer_type& get_type() const;

Returns the type of the specified pointer.

os_pointer_literal::set name()

void set_name(const char*);

Sets the name of the specified literal.

os_pointer_literal::set_type()

void set_type(os_pointer_type&);

Sets the type of the specified pointer.

os_pointer_to_member_type

class os_pointer_to_member_type : public os_pointer_type

This class is part of the ObjectStore metaobject protocol, which provides access to ObjectStore schemas. An instance of this class represents a C++ pointer-to-member type. This class is derived from **os_pointer_type**.

os_pointer_to_member-type::create()

static os_pointer_to_member_type& create(os_type* target,
os_class_type*);

The argument is used to initialize **target_class** and **target_type**.

os_pointer_to_member_type::get_target_class()

const os_class_type &get_target_class() const;

Returns the class associated with the specified pointer-to-member.

os_pointer_to_member_type::set_target_class()

void set_target_class(os_class_type&);

Sets the class associated with the specified pointer-to-member.

os_pointer_type

class os_pointer_type : public os_type

This class is part of the ObjectStore metaobject protocol, which provides access to ObjectStore schemas. An instance of this class represents a C++ pointer type. This class is derived from **os_type**.

os_pointer_type::create()

static os_pointer_type& create(os_type* target);

The argument is used to initialize the attribute **target_type**.

os_pointer_type::get_target_type()

const os_type &get_target_type() const;

Returns the type of object pointed to by instances of the specified pointer type.

os_pointer_type::set_target_type()

void set_target_type(os_type&);

Sets the type of object pointed to by instances of the specified pointer type.

os_pragma

This class is part of the ObjectStore metaobject protocol, which provides access to ObjectStore schemas. An instance of this class represents a C++ pragma.

os_pragma::create()

static os_pragma& create(const char*);

Creates a new pragma and associates the specified string with it.

os_pragma::get_string()

const char *get_string() const;

Returns the string associated with the specified pragma.

os_pragma::is_recognized()

os_boolean is_recognized() const;

Returns nonzero if the specified pragma is recognized.

os_pvar

When a pointer to persistent memory is assigned to a transiently allocated variable, the value of the variable is valid only until the end of the transaction in which the assignment was made. Using database entry points typically involves looking up a root and retrieving its value — a pointer to the entry point. Frequently this pointer is assigned to a transiently allocated variable for future use. But its use is limited, since it normally will not remain valid in subsequent transactions.

One way to deal with this situation is to re-retrieve the pointer in each subsequent transaction in which it is required. However, a convenient alternative is provided by ObjectStore *pvars*. These allow you to maintain, across transactions, a valid pointer to an entry-point object.

To use pvars, you define the variable you want to hold the pointer to the entry point. Then you pass the variable's address to the function **os_pvar::os_pvar()**, along with the name of the root that points to the desired entry-point object, and a pointer to the database containing the root.

This function is the constructor for the class **os_pvar**, but you never have to explicitly use the instance of **os_pvar** that results. Once you have called this function, ObjectStore automatically maintains an association between the variable and the entry point. At the beginning of each transaction in the current process, if the database containing the specified root is open, ObjectStore establishes a valid pointer to the entry-point object as the value of the variable. It also sets the variable to point to the entry point when the database becomes open during a transaction.

Instances of **os_pvar** must be allocated on the stack, not the heap, so do not create **os_pvar**s with **operator new()**.

As with **os_database_root::get_value()**, you can also supply an **os_typespec*** to **os_pvar::os_pvar()**, for additional type safety. ObjectStore will check that the specified typespec matches the typespec stored with the root. Note that it checks only that the typespec supplied matches the stored typespec, and does not check the type of the entry-point object itself.

Note that, even though you can use this variable from one transaction to the next without re-retrieving its value, you cannot use it *between* transactions. As always, you must be within a transaction to access persistent data. Between transactions, ObjectStore automatically sets the variable to **0**. The variable is also set to **0** during a transaction if the database containing its associated root is closed.

Note also that you should not try to set the value of this variable, since ObjectStore handles all assignments of values to it.

You can also create an entry point and root using **os_pvar::os_pvar()** by supplying a pointer to an initialization function. The function should allocate the entry-point object in a given database and return a pointer to the new object.

This function will be executed upon the call to **os_pvar()** or at the beginning of subsequent transactions, if the database to contain the root is open and ObjectStore cannot find the specified root in that database. It will also be called when this database becomes open during a transaction and ObjectStore cannot find the root in that database.

The predefined functions **os_pvar::init_pointer()**, **os_pvar::init_int()**, and **os_pvar::init_long()** can be used as initialization functions. They allocate pointers, **int**s, and **long**s, respectively, and initialize them to **0**.

os_pvar::os_pvar()

```
os_pvar(
    os_database_p db,
    os_void_p location,
    os_char_p root_name,
    os_typespec *typespec = 0,
    os_void_p(*init_fn)(os_database*) = 0
);
```

Constructs an **os_pvar**. **db** points to the database containing the pvar’s associated database root. **location** is the address of the variable whose value is to be maintained across transactions. **root_name** is the name of the associated root. **typespec** is the typespec stored with the associated root. **init_fn** is a pointer to an initialization function.

Instances of **os_pvar** must be allocated on the stack, not the heap, so do not create **os_pvar**s with **operator new()**.

os_pvar::~os_pvar()

~os_pvar();

Breaks the association between a pvar's location and database root.

os_pvar::init_pointer()

static void *init_pointer(os_database *db);

Allocates a pointer, initializes it to **0**, and returns a pointer to the allocated pointer.

os_pvar::init_int()

static void *init_int(os_database *db);

Allocates an **int**, initializes it to **0**, and returns a pointer to the allocated **int**.

os_pvar::init_long()

static void *init_long(os_database *db);

Allocates a **long**, initializes it to **0**, and returns a pointer to the allocated **long**.

os_rawfs_entry

The functions **os_dbutil:stat()** and **os_dbutil::list_directory()** return pointers to instances of this class. Each **os_rawfs_entry** represents a rawfs directory, database, or link.

os_rawfs_entry::get_abs_path()

const char *get_abs_path() const;

Returns the absolute pathname for the specified entry.

os_rawfs_entry::get_creation_time()

os_unixtime_t get_creation_time() const;

Returns the creation time for the specified entry.

os_rawfs_entry::get_group_name()

const char *get_group_name() const;

Returns the name of the primary group for the specified entry.

os_rawfs_entry::get_link_host()

const char *get_link_host() const;

Returns the name of the host for the target of the link represented by the specified entry. If the entry does not represent a link, **0** is returned.

os_rawfs_entry::get_link_path()

const char *get_link_path() const;

Returns the pathname of the target of the link represented by the specified entry. If the entry does not represent a link, **0** is returned.

os_rawfs_entry::get_n_sectors()

os_unsigned_int32 get_n_sectors() const;

Returns the number of sectors in the specified entry.

os_rawfs_entry::get_name()

const char *get_name() const;

Returns the terminal component of the pathname for the specified entry.

os_rawfs_entry::get_permission()

os_unsigned_int32 get_permission() const;

Returns a bit-wise disjunction representing the permissions on the specified entry.

os_rawfs_entry::get_server_host()

const char *get_server_host() const;

Returns the name of the host for the specified entry.

os_rawfs_entry::get_type()

os_int32 get_type() const;

Returns an enumerator indicating whether the specified entry represents a directory, database, or link. One of the following enumerators is returned:

- **os_rawfs_entry::OSU_DIRECTORY**
- **os_rawfs_entry::OSU_DATABASE**
- **os_rawfs_entry::OSU_LINK**

os_rawfs_entry::get_user_name()

const char *get_user_name() const;

Returns the name of the user for the specified entry.

os_rawfs_entry::is_db()

os_boolean is_db() const;

Returns nonzero if the specified entry represents a database; returns **0** otherwise.

os_rawfs_entry::is_dir()

os_boolean is_dir() const;

Returns nonzero if the specified entry represents a directory; returns **0** otherwise.

os_rawfs_entry::is_link()

os_boolean is_link() const;

Returns nonzero if the specified entry represents a link; returns **0** otherwise.

os_rawfs_entry::operator =()

os_rawfs_entry &operator =(const os_rawfs_entry&);

Modifies the left operand so that it is a copy of the right operand. The copy behaves just like the original with respect to the member functions of **os_rawfs_entry**.

os_rawfs_entry::OSU_DATABASE

Enumerator used as possible return value for **os_rawfs_entry::get_type()**, indicating that the specified **os_rawfs_entry** represents a database.

os_rawfs_entry::OSU_DIRECTORY

Enumerator used as possible return value for **os_rawfs_entry::get_type()**, indicating that the specified **os_rawfs_entry** represents a directory.

os_rawfs_entry::OSU_LINK

Enumerator used as possible return value for **os_rawfs_entry::get_type()**, indicating that the specified **os_rawfs_entry** represents a link.

os_rawfs_entry::os_rawfs_entry()

os_rawfs_entry(const os_rawfs_entry&);

Creates an **os_rawfs_entry** that is a copy of the specified **os_rawfs_entry**. The copy behaves just like the original with respect to the member functions of the class **os_rawfs_entry**.

os_rawfs_entry::~os_rawfs_entry()

~os_rawfs_entry();

Frees storage associated with the specified **os_rawfs_entry**.

os_real_type

class os_real_type : public os_type

This class is part of the ObjectStore metaobject protocol, which provides access to ObjectStore schemas. An instance of this class represents a C++ floating type. This class is derived from **os_type**.

Programs using this class must include **<ostore/ostore.hh>**, followed by **<ostore/coll.hh>** (if used), followed by **<ostore/mop.hh>**.

os_real_type::create()

static os_real_type &create(const char*);

Creates an **os_real_type** representing the type with the specified name.

os_real_type::create_float()

static os_real_type &create_float();

Creates an **os_real_type** representing the type **float**.

os_real_type::create_double()

static os_real_type &create_double();

Creates an **os_real_type** representing the type **double**.

os_real_type::create_long_double()

static os_real_type &create_long_double();

Creates an **os_real_type** representing the type **long double**.

os_Reference

Instances of the class **os_Reference** can be used as substitutes for cross-database and cross-transaction pointers. References are valid under a wider array of circumstances than are pointers to persistent storage.

A pointer to persistent storage assigned to transient memory is valid only until the end of the outermost transaction in which the assignment occurred, unless **objectstore::retain_persistent_ addresses()** is used. In addition, a pointer to storage in one database assigned to storage in another database is valid only until the end of the outermost transaction in which the assignment occurred, unless **os_database::allow_external_pointers()** or **os_ segment::allow_external_pointers()** is used.

os_References, in contrast, are always valid across transaction boundaries, as well as across databases.

Once the object referred to by a reference is deleted, use of the reference accesses arbitrary data and might cause a segmentation violation. But see **os_Reference_protected** on page 224.

The class **os_Reference** is *parameterized,* with a parameter for indicating the type of the object referred to by a reference. This means that when specifying **os_Reference** as a function's formal parameter, or as the type of a variable or data member, you must specify the parameter — the reference's *referent type*. You do this by appending to **os_Reference** the name of the referent type enclosed in angle brackets (< >):

os_Reference<*referent-type-name*>

The referent type must be a class. For references to built-in types such as **int** and **char** see **os_reference** on page 209.

The referent type parameter, **T**, occurs in the signatures of some of the functions described below. The parameter is used by the compiler to detect type errors.

You can create a reference to serve as substitute for a pointer of type **T*** by initializing a variable of type **os_Reference<T>** with a **T***, or by assigning a **T*** to a variable of type **os_Reference<T>** (implicitly invoking the conversion constructor **os_ Reference::os_Reference(T*)**).

```
part *a_part; ...
os_Reference<part> part_ref = a_part;
```

Usually, when an **os_Reference<T*>** is used where a **T*** is expected, **os_Reference::operator –>()** or **os_Reference::operator T*()** is implicitly invoked, returning a valid pointer to the object referred to by the **os_Reference**.

```
printf("%d\n", part_ref->part_id);
```

Not all C++ operators have special reference class overloadings. References do not behave like pointers in the context of **[]** and **++**, for example.

In some cases involving multiple inheritance, comparing two references has a different result from comparing the corresponding pointers. For example, for **==** comparisons, if the referent type of one operand is a nonleftmost base class of the referent type of the other operand, the result is always 1.

Each instance of this class stores a relative pathname to identify the referent database. The pathname is relative to the lowest common directory in the pathnames of the referent database and the database containing the reference. For example, if a reference stored in **/A/B/C/db1** refers to data in **/A/B/D/db2**, the lowest common directory is **A/B**, so the relative pathname **../../D/db2** is used.

This means that if you copy a database containing a reference, the reference in the copy and the reference in the original might refer to different databases. To change the database a reference refers to, you can use the ObjectStore utility **oschangedbref**. See *ObjectStore Management*.

Using memcpy() with persistent os_References

You can use the C++ **memcpy()** function to copy a persistent **os_Reference** only if the target object is in the same segment as the source object. This is because all persistent **os_Reference**s use **os_segment::of(this)** for **os_Reference** resolution processing and the resoultion will be incorrect if the **os_Reference** has been copied to a different segment.

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.

All ObjectStore programs must include the header file **<ostore/ostore.hh>**.

os_Reference::dump()

char *dump(const char *db_str) const;

Returns a heap-allocated text string representing the specified reference. However, unlike the string returned by the **char * os_ Reference::dump(void)** method, this string does not contain an absolute database path. The returned string is intended to be used as the **dump_str** parameter of an **os_Reference** load method of the form **load(const char* dump_str, os_database* db)**. It is the responsibility of the caller of **load** to ensure that the **db** parameter passed to the load method is the same as the database of the dumped reference. It is the user's responsibility to delete the returned string when finished using the string.

This operation is useful in those applications in which you do not want the overhead of storing the absolute database path in the dumped strings.

os_Reference::get_database()

os_database *get_database() const;

Returns a pointer to the database containing the object referred to by the specified reference.

os_Reference::get_database_key()

char* get_database_key(const char* dump_str);

Returns a heap-allocated string containing the **database_key** component of the string **dump_str**. **dump_str** must have been generated using the **dump** operation. Otherwise, the exception err_reference_syntax is raised. It is the user's responsibility to delete the returned string when finished using the string.

os_Reference::get_open_database()

os_database *get_open_database() const;

Returns a pointer to the database containing the object referred to by the specified **os_Reference**. Opens the database.

os_Reference::get_os_typespec()

static os_typespec *get_os_typespec();

Returns an **os_typespec*** for the class **os_Reference**.

os_Reference::hash()

os_unsigned_int32 hash() const;

Returns an integer suitable for use as a hash table key. The value returned is always the same for a reference to a given referent.

os_Reference::load()

void load(const char* dump_str, const os_database* db);

The **dump_str** parameter is assumed to be the result of a call to a compatible **os_Reference** dump method. It is the responsibility of the caller of **load** to ensure that the **db** parameter passed to the load method is the same as the database of the originally dumped reference.

The loaded reference refers to the same object as the **os_Reference** used to dump the string as long as the **db** parameter is the same as the database of the dumped reference.

The exception err_reference_syntax is raised if the **dump_str** is not in the expected format or if the **dump_str** was dumped from a protected reference.

os_Reference::operator T*()

operator T*() const;

Returns the valid **T*** for which the specified reference is a substitute.

os_Reference::operator –>()

T* operator –>() const;

Returns the valid **T*** for which the specified reference is a substitute.

os_Reference::operator =()

os_Reference<T> &operator=(const os_Reference<T>&);

Establishes the referent of the right operand as the referent of the left operand.

os_Reference<T> &operator=(const T*);

Establishes the object pointed to by the right operand as the referent of the left operand.

os_Reference::operator ==()

os_boolean operator ==(os_Reference const&) const;

Returns **1** if the arguments have the same referent; returns **0** otherwise.

os_Reference::operator !()

os_boolean operator !(os_Reference const&) const;

Returns **1** if the **os_Reference** argument is pointing to NULL; returns **0** otherwise.

os_Reference::operator !=()

os_boolean operator !=(os_Reference const&) const;

Returns **1** if the arguments have different referents; returns **0** otherwise.

os_Reference::operator <()

os_boolean operator <(os_Reference const&) const;

If the first argument and second argument refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument precedes the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

os_Reference:operator >()

os_boolean operator >(os_Reference const&) const;

If the first argument and second argument refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument follows the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

os_Reference:operator >=()

os_boolean operator >=(os_Reference const&) const;

If the first argument and second argument refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument follows or is the same as the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

os_Reference:operator <=()

os_boolean operator <=(os_Reference const&) const;

If the first argument and second argument refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument precedes or is the same as the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

os_Reference::os_Reference()

os_Reference(T*);

Constructs a reference to substitute for the specified **T***.

os_Reference::resolve()

T *resolve() const;

Returns the valid **T*** for which the specified reference is a substitute.

os_reference

Instances of the class **os_reference** can be used as substitutes for cross-database and cross-transaction pointers. References are valid under a wider array of circumstances than are pointers to persistent storage.

A pointer to persistent storage assigned to transient memory is valid only until the end of the outermost transaction in which the assignment occurred, unless **objectstore::retain_persistent_ addresses()** is used. In addition, a pointer to storage in one database assigned to storage in another database is valid only until the end of the outermost transaction in which the assignment occurred, unless **os_database::allow_external_pointers()** or **os_ segment::allow_external_pointers()** is used.

os_references, in contrast, are always valid across transaction boundaries, as well as across databases.

Once the object referred to by a reference is deleted, use of the reference accesses arbitrary data and might cause a segmentation violation. But see **os_reference_protected** on page 231.

You can create a reference to serve as substitute for a pointer by initializing a variable of type **os_reference** with the pointer, or by assigning the pointer to a variable of type **os_reference** (implicitly invoking the conversion constructor **os_reference::os_ reference(void*)**). In general, a pointer can be used anywhere an **os_reference** is expected, and the conversion constructor will be invoked.

```
part *a_part = ... ;
os_reference part_ref = a_part;
```

When an **os_reference** is cast to pointer-to-referent-type (that is, pointer to the type of object referred to by the reference), **os_ reference::operator void*()** is implicitly invoked, returning a valid pointer to the object referred to by the **os_reference**.

```
printf("%d\n", (part*)(part_ref)->part_id);
```

In some cases involving multiple inheritance, comparing two references has a different result from comparing the corresponding pointers. For example, for **==** comparisons, if the referent type of one operand is a nonleftmost base class of the referent type of the other operand, the result is always 1. This is because comparing references never results in the pointer adjustment described in Section 10.3c of the *C++ Annotated Reference Manual*.

Each instance of this class stores a relative pathname to identify the referent database. The pathname is relative to the lowest common directory in the pathnames of the referent database and the database containing the reference. For example, if a reference stored in **/A/B/C/db1** refers to data in **/A/B/D/db2**, the lowest common directory is **A/B**, so the relative pathname **../../D/db2** is used.

This means that if you copy a database containing a reference, the reference in the copy and the reference in the original might refer to different databases. To change the database a reference refers to, you can use the ObjectStore utility **oschangedbref**. See *ObjectStore Management*.

Using memcpy() with persistent os_references

You can use the C++ **memcpy()** function to copy a persistent **os_reference** only if the target object is in the same segment as the source object. This is because all persistent **os_reference**s use **os_segment::of(this)** for **os_reference** resolution processing and the resolution will be incorrect if the **os_reference** has been copied to a different segment.

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.

All ObjectStore programs must include the header file **<ostore/ostore.hh>**.

os_reference::dump()

char* dump(const char* db_str) const;

Returns a heap-allocated text string representing the specified reference. However, unlike the string returned by the **char * os_reference::dump(void)** method, this string does not contain an absolute database path. The returned string is intended to be used as the **dump_str** parameter of an **os_reference** load method of the form **load(const char* dump_str, os_database* db)**. It is the responsibility of the caller of **load** to ensure that the **db** parameter passed to the load method is the same as the database of the dumped reference. It is the user’s responsibility to delete the returned string when finished using the string.

This operation is useful in those applications in which you do not want the overhead of storing the absolute database path in the dumped strings.

os_reference::get_database()

os_database *get_database() const;

Returns a pointer to the database containing the object referred to by the specified reference.

os_reference::get_database_key()

char* get_database_key(const char* dump_str);

Returns a heap-allocated string containing the **database_key** component of the string **dump_str**. **dump_str** must have been generated using the **dump** operation. Otherwise, the exception err_reference_syntax is raised. It is the user’s responsibility to delete the returned string when finished using the string.

os_reference::get_open_database()

os_database *get_open_database() const;

Returns a pointer to the database containing the object referred to by the specified **os_reference**. Opens the database.

os_reference::get_os_typespec()

static os_typespec *get_os_typespec();

Returns an **os_typespec*** for the class **os_reference**.

os_reference::hash()

os_unsigned_int32 hash() const;

Returns an integer suitable for use as a hash table key. The value returned is always the same for an **os_reference** to a given referent.

os_reference::load()

void load(const char*);

If the specified **char*** points to a string generated from a reference with **os_reference::dump()**, calling this function makes the specified reference refer to the same object referred to by the reference used to generate the string.

void load(const char* dump_str, const os_database* db);

The **dump_str** parameter is assumed to be the result of a call to a compatible **os_Reference** dump method. It is the responsibility of the caller of **load** to ensure that the **db** parameter passed to the load method is the same as the database of the originally dumped reference.

The loaded reference refers to the same object as the **os_Reference** used to dump the string as long as the **db** parameter is the same as the database of the dumped reference.

The exception err_reference_syntax is raised if the **dump_str** is not in the expected format or if the **dump_str** was dumped from a protected reference.

os_reference::operator void*()

operator void*() const;

Returns the valid pointer for which the specified reference is a substitute.

os_reference::operator =()

os_reference &operator=(const os_reference&);

Establishes the referent of the right operand as the referent of the left operand.

os_reference &operator=(const void*);

Establishes the object pointed to by the right operand as the referent of the left operand.

os_reference::operator ==()

os_boolean operator ==(os_reference const&) const;

Returns **1** if the arguments have the same referent; returns **0** otherwise.

os_reference::operator !=()

os_boolean operator !=(os_reference const&) const;

Returns **1** if the arguments have different referents; returns **0** otherwise.

os_reference::operator <()

os_boolean operator <(os_reference const&) const;

If the first and second arguments refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument precedes the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

os_reference::operator >()

os_boolean operator >(os_reference const&) const;

If the first and second arguments refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument follows the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

os_reference::operator >=()

os_boolean operator >=(os_reference const&) const;

If the first and second arguments refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument follows or is the same as the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

os_reference::operator <=()

os_boolean operator <=(os_reference const&) const;

If the first and second arguments refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument precedes or is the same as the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

os_reference::os_reference()

os_reference(const void*);

Constructs a reference to substitute for the specified **void***.

os_reference::resolve()

void* resolve() const;

Returns the valid **void*** for which the specified reference is a substitute.

os_Reference_local

Instances of the class **os_Reference_local** can be used as substitutes for cross-database and cross-transaction pointers. References are valid under a wider array of circumstances than are pointers to persistent storage.

A pointer to persistent storage assigned to transient memory is valid only until the end of the outermost transaction in which the assignment occurred, unless **objectstore::retain_persistent_ addresses()** is used. In addition, a pointer to storage in one database assigned to storage in another database is valid only until the end of the outermost transaction in which the assignment occurred, unless **os_database::allow_external_pointers()** or **os_ segment::allow_external_pointers()** is used.

os_Reference_locals, in contrast, are always valid across transaction boundaries, as well as across databases.

An **os_Reference_local** is smaller than an **os_Reference**, but resolving it requires explicit specification of the referent database.

Once the object referred to by a reference is deleted, use of the reference accesses arbitrary data and might cause a segmentation violation. But see **os_Reference_protected_local** on page 237.

The class **os_Reference_local** is *parameterized*, with a parameter for indicating the type of the object referred to by a reference. This means that when specifying **os_Reference_local** as a function's formal parameter, or as the type of a variable or data member, you must specify the parameter — the reference's *referent type*. You do this by appending to **os_Reference_local** the name of the referent type enclosed in angle brackets (< >):

os_Reference_local<*referent-type-name*>

The referent type must be a class. For local references to built-in types, such as **int** and **char**, see **os_reference_local** on page 220.

The referent type parameter, **T**, occurs in the signatures of some of the functions described below. The parameter is used by the compiler to detect type errors.

You can create a reference to serve as substitute for a pointer of type **T*** by initializing a variable of type **os_Reference_local<T>**

with a **T***, or by assigning a **T*** to a variable of type **os_Reference_local<T>** (implicitly invoking the conversion constructor **os_Reference_local::os_Reference_local(T*)**).

```
part *a_part = ... ;
os_Reference_local<part> part_ref = a_part;
```

When the member function **resolve()** is applied to an **os_Reference_local**, with a pointer to the referent database as argument, a valid pointer to the referent object is returned.

```
printf("%d\n", part_ref.resolve(db1)->part_id);
```

In some cases involving multiple inheritance, comparing two references has a different result from comparing the corresponding pointers. For example, for **==** comparisons, if the referent type of one operand is a nonleftmost base class of the referent type of the other operand, the result is always 1.

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.

All ObjectStore programs must include the header file **<ostore/ostore.hh>**.

os_Reference_local::dump()

char *dump(const char *database_name) const;

Returns a heap-allocated text string representing the specified reference. When this string is passed to **os_Reference_local::load()**, the result is a reference to the same object referred to by the dumped reference. It is the user's responsibility to delete the returned string.

os_Reference_local::get_database_key()

char* get_database_key(const char* dump_str);

Returns a heap-allocated string containing the **database_key** component of the string **dump_str**. **dump_str** must have been generated using the **dump** operation. Otherwise, the exception err_reference_syntax is raised. It is the user's responsibility to delete the returned string when finished using the string.

os_Reference_local::get_os_typespec()

static os_typespec *get_os_typespec();

Returns an **os_typespec*** for the class **os_Reference_local**.

os_Reference_local::hash()

os_unsigned_int32 hash() const;

Returns an integer suitable for use as a hash table key. The value returned is always the same for an **os_Reference_local** to a given referent.

os_Reference_local::load()

void load(const char*);

If the specified **char*** points to a string generated from a reference with **os_Reference_local::dump()**, calling this function makes the specified reference refer to the same object referred to by the reference used to generate the string.

os_Reference_local::operator =()

os_Reference_local<T> &operator=(const os_Reference_local<T>&);

Establishes the referent of the right operand as the referent of the left operand.

os_Reference_local<T> &operator=(const T*);

Establishes the object pointed to by the right operand as the referent of the left operand.

os_Reference_local::operator ==()

os_boolean operator ==(os_Reference_local const&) const;

Returns **1** if the arguments have the same referent; returns **0** otherwise.

os_Reference_local::operator !=()

os_boolean operator !=(os_Reference_local const&) const;

Returns **1** if the arguments have different referents; returns **0** otherwise.

os_Reference_local::operator <()

os_boolean operator <(os_Reference_local const&) const;

If the first argument and second argument refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument precedes the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

os_Reference_local::operator >()

os_boolean operator >(os_Reference_local const&) const;

If the first argument and second argument refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument follows the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

os_Reference_local::operator >=()

os_boolean operator >=(os_Reference_local const&) const;

If the first argument and second argument refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument follows or is the same as the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

os_Reference_local::operator <=()

os_boolean operator <=(os_Reference_local const&) const;

If the first argument and second argument refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument precedes or is the same as the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

os_Reference_local::os_Reference_local()

os_Reference_local(T*);

Constructs a reference to substitute for the specified **T***.

os_Reference_local::resolve()

T* resolve(const os_database*) const;

Returns the valid **T*** for which the specified reference is a substitute. The database containing the storage pointed to by the **T*** must be specified.

os_reference_local

Instances of the class **os_reference_local** can be used as substitutes for cross-database and cross-transaction pointers. References are valid under a wider array of circumstances than are pointers to persistent storage.

A pointer to persistent storage assigned to transient memory is valid only until the end of the outermost transaction in which the assignment occurred, unless **objectstore::retain_persistent_ addresses()** is used. In addition, a pointer to storage in one database assigned to storage in another database is valid only until the end of the outermost transaction in which the assignment occurred, unless **os_database::allow_external_pointers()** or **os_ segment::allow_external_pointers()** is used.

os_reference_locals, in contrast, are always valid across transaction boundaries, as well as across databases.

An **os_reference_local** is smaller than an **os_reference**, but resolving it requires explicit specification of the referent database.

Once the object referred to by a reference is deleted, use of the reference accesses arbitrary data and might cause a segmentation violation. But see **os_reference_protected_local** on page 242.

You can create a reference to serve as substitute for a pointer by initializing a variable of type **os_reference_local** with the pointer, or by assigning the pointer to a variable of type **os_reference_local** (implicitly invoking the conversion constructor **os_reference_ local::os_reference_local(void*)**). In general, a pointer can be used anywhere an **os_reference_local** is expected, and the conversion constructor will be invoked.

```
part *a_part = ... ;
os_reference_local part_ref = a_part;
```

When the member function **resolve()** is applied to an **os_ reference_local**, with a pointer to the referent database as argument, a valid pointer to the referent object is returned.

```
printf("%d\n", part_ref.resolve(db1)->part_id);
```

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.

All ObjectStore programs must include the header file **<ostore/ostore.hh>**.

os_reference_local::dump()

char *dump(const char *database_name) const;

Returns a heap-allocated text string representing the specified reference. When this string is passed to **os_reference_local::load()**, the result is a reference to the same object referred to by the dumped reference. It is the user's responsibility to delete the returned string.

os_reference_local::get_database_key()

char* get_database_key(const char* dump_str);

Returns a heap-allocated string containing the **database_key** component of the string **dump_str**. **dump_str** must have been generated using the **dump** operation. Otherwise, the exception err_reference_syntax is raised. It is the user's responsibility to delete the returned string when finished using the string.

os_reference_local::get_os_typespec()

static os_typespec *get_os_typespec();

Returns an **os_typespec*** for the class **os_reference_local**.

os_reference_local::hash()

os_unsigned_int32 hash() const;

Returns an integer suitable for use as a hash table key. The value returned is always the same for an **os_reference_local** to a given referent.

os_reference_local::load()

void load(const char*);

If the specified **char*** points to a string generated from a reference with **os_reference_local::dump()**, calling this function makes the specified reference refer to the same object referred to by the reference used to generate the string.

os_reference_local::operator =()

os_reference_local &operator=(const os_reference_local&);

Establishes the referent of the right operand as the referent of the left operand.

os_reference_local &operator=(const void*);

Establishes the object pointed to by the right operand as the referent of the left operand.

os_reference_local::operator ==()

os_boolean operator ==(os_reference_local const&) const;

Returns **1** if the arguments have the same referent; returns **0** otherwise.

os_reference_local::operator !=()

os_boolean operator !=(os_reference_local const&) const;

Returns **1** if the arguments have different referents; returns **0** otherwise.

os_reference_local::operator <()

os_boolean operator <(os_reference_local const&) const;

If the first and second arguments refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument precedes the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

os_reference_local::operator >()

os_boolean operator >(os_reference_local const&) const;

If the first and second arguments refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument follows the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

os_reference_local::operator >=()

os_boolean operator >=(os_reference_local const&) const;

If the first and second arguments refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument follows or is the same as the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

os_reference_local::operator <=()

os_boolean operator <=(os_reference_local const&) const;

If the first and second arguments refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument precedes or is the same as the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

os_reference_local::os_reference_local()

os_reference_local(const void*);

Constructs a reference to substitute for the specified **void***.

os_reference_local::resolve()

void *resolve(const os_database*) const;

Returns the valid **void*** for which the specified reference is a substitute. The database containing the storage pointed to by the **void*** must be specified.

os_Reference_protected

Instances of the class **os_Reference_protected** can be used as substitutes for cross-database and cross-transaction pointers. References are valid under a wider array of circumstances than are pointers to persistent storage.

A pointer to persistent storage assigned to transient memory is valid only until the end of the outermost transaction in which the assignment occurred, unless **objectstore::retain_persistent_ addresses()** is used. In addition, a pointer to storage in one database assigned to storage in another database is valid only until the end of the outermost transaction in which the assignment occurred, unless **os_database::allow_external_pointers()** or **os_ segment::allow_external_pointers()** is used.

os_Reference_protecteds, in contrast, are always valid across transaction boundaries, as well as across databases.

Once the object referred to by an **os_Reference_protected** is deleted, use of the **os_Reference_protected** causes an err_ reference_not_found exception to be signaled. If the referent database has been deleted, err_database_not_found is signaled.

The class **os_Reference_protected** is *parameterized*, with a parameter for indicating the type of the object referred to by a reference. This means that when specifying **os_Reference_ protected** as a function's formal parameter, or as the type of a variable or data member, you must specify the parameter — the reference's *referent type*. You do this by appending to **os_ Reference_protected** the name of the referent type enclosed in angle brackets (< >):

os_Reference_protected<*referent-type-name*>

The referent type must be a class. For protected references to built-in types, such as **int** and **char**, see **os_reference_protected** on page 231.

The referent type parameter, **T**, occurs in the signatures of some of the functions described below. The parameter is used by the compiler to detect type errors.

You can create a reference to serve as substitute for a pointer of type **T*** by initializing a variable of type **os_Reference_**

protected<T> with a **T***, or by assigning a **T*** to a variable of type **os_Reference_protected<T>** (implicitly invoking the conversion constructor **os_Reference_protected::os_Reference_ protected(T*)**). This **T*** must not point to transient memory.

```
part *a_part = ... ;
os_Reference_protected<part> part_ref = a_part;
```

When an **os_Reference_protected<T*>** is used where a **T*** is expected, **os_Reference_protected::operator ->()** or **os_Reference_ protected::operator T*()** is implicitly invoked, returning a valid pointer to the object referred to by the **os_Reference_protected**.

```
printf("%d\n", part_ref->part_id);
```

Not all C++ operators have special reference class overloadings. References do not behave like pointers in the context of **[]** and **++**, for example.

In some cases involving multiple inheritance, comparing two references has a different result from comparing the corresponding pointers. For example, for **==** comparisons, if the referent type of one operand is a nonleftmost base class of the referent type of the other operand, the result is always 1. This is because comparing references never results in the pointer adjustment described in Section 10.3c of the *C++ Annotated Reference Manual*.

Each instance of this class stores a relative pathname to identify the referent database. The pathname is relative to the lowest common directory in the pathnames of the referent database and the database containing the reference. For example, if a reference stored in **/A/B/C/db1** refers to data in **/A/B/D/db2**, the lowest common directory is **A/B**, so the relative pathname **../../Dfs/db2** is used.

This means that if you copy a database containing a reference, the reference in the copy and the reference in the original might refer to different databases. To change the database a reference refers to, you can use the ObjectStore utility **oschangedbref**. See *ObjectStore Management*.

Using memcpy() with persistent os_Reference_protecteds

You can use the C++ **memcpy()** function to copy a persistent **os_ Reference_protected** only if the target object is in the same segment as the source object. This is because all persistent objects of the type **os_Reference_protected** use **os_segment::of(this)** for reference resolution processing and the resolution will be incorrect if the **os_Reference_protected** has been copied to a different segment.

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.

All ObjectStore programs must include the header file **<ostore/ostore.hh>**.

os_Reference_protected::deleted()

os_boolean deleted() const;

Returns **1** (true) if the object to which the specified reference refers has been deleted; returns **0** (false) otherwise.

os_Reference_protected::dump()

char* dump(const char* db_str) const;

Returns a heap-allocated string for the reference that the method was called. However, unlike the string returned by the **char* os_ Reference_protected::dump(void)** method, the returned string does not contain an absolute database pathname. The returned string is intended to be used as the **dump_str** parameter of an **os_ Reference_protected** load method of the form **load(const char* dump_str, os_database* db)**. It is the responsibility of the caller of **load** to ensure that the **db** parameter passed to the load method is the same as the database of the dumped reference. It is the user's responsibility to delete the returned string when finished using the string.

This operation is useful in those applications in which you do not want the overhead of storing the absolute database path in the dumped strings.

os_Reference_protected::forget()

void forget();

Frees the memory in the underlying table used to associate the specified **os_Reference_protected** with its referent. Subsequent use of the **os_Reference_protected** will result in a run-time error.

os_Reference_protected::get_database()

os_database *get_database() const;

Returns a pointer to the database containing the object referred to by the specified reference.

os_Reference_protected::get_database_key()

char* get_database_key(const char* dump_str);

Returns a heap-allocated string containing the **database_key** component of the string **dump_str**. **dump_str** must have been generated using the **dump** operation. Otherwise, the exception err_reference_syntax is raised. It is the user's responsibility to delete the returned string when finished using the string.

os_Reference_protected::get_open_database()

os_database *get_open_database() const;

Returns a pointer to the database containing the object referred to by the specified **os_Reference_protected**. Opens the database.

os_Reference_protected::get_os_typespec()

static os_typespec *get_os_typespec();

Returns an **os_typespec*** for the class **os_Reference_protected**.

os_Reference_protected::hash()

os_unsigned_int32 hash() const;

Returns an integer suitable for use as a hash table key. The value returned is always the same for an **os_Reference_protected** to a given referent.

os_Reference_protected::load()

void load(const char* dump_str, const os_database* db);

The **dump_str** parameter is assumed to be the result of a call to a compatible **os_Reference dump** method. It is the responsibility of the caller of **load** to ensure that the **db** parameter passed to the load method is the same as the database of the originally dumped reference. The loaded reference refers to the same object as the **os_Reference** used to dump the string as long as the **db** parameter is the same as the database of the dumped reference. The exception err_reference_syntax is raised if the **dump_str** is not in the expected format or if the **dump_str** was dumped from a nonprotected reference.

os_Reference_protected::operator T*()

operator T*() const;

Returns the **T*** for which the specified **os_Reference_protected** is a substitute. If the referent has been deleted, err_reference_not_found is signaled. If the referent database has been deleted, err_database_not_found is signaled.

os_Reference_protected::operator –>()

T* operator –>() const;

Returns the **T*** for which the specified **os_Reference_protected** is a substitute. If the referent has been deleted, err_reference_not_found is signaled. If the referent database has been deleted, err_database_not_found is signaled.

os_Reference_protected::operator =()

```
os_Reference_protected<T> &operator=(
    const os_Reference_protected<T>&
);
```

Establishes the referent of the right operand as the referent of the left operand.

os_Reference_protected<T> &operator=(const T*);

Establishes the object pointed to by the right operand as the referent of the left operand.

os_Reference_protected::operator ==()

os_boolean operator ==(os_Reference_protected const&) const;

Returns **1** if the arguments have the same referent; returns **0** otherwise.

os_Reference_protected::operator !()

os_boolean operator !() const;

Returns nonzero if the reference has no current referent.

os_Reference_protected::operator !=()

os_boolean operator !=(os_Reference_protected const&) const;

Returns **1** if the arguments have different referents; returns **0** otherwise.

os_Reference_protected::operator <()

os_boolean operator <(os_Reference_protected const&) const;

If the first argument and second argument refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument precedes the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

os_Reference_protected::operator >()

os_boolean operator >(os_Reference_protected const&) const;

If the first argument and second argument refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument follows the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

os_Reference_protected::operator >=()

os_boolean operator >=(os_Reference_protected const&) const;

If the first argument and second argument refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument follows or is the same as the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

os_Reference_protected::operator <=()

os_boolean operator <=(os_Reference_protected const&) const;

If the first argument and second argument refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument precedes or is the same as the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

os_Reference_protected::os_Reference_protected()

os_Reference_protected(T*);

Constructs an **os_Reference_protected** to substitute for the specified **T***. If the **T*** points to transient memory, err_reference_to_transient is signaled. **0** is a legal argument.

os_Reference_protected::resolve()

T *resolve() const;

Returns the valid **T*** for which the specified reference is a substitute. If the referent has been deleted, err_reference_not_found is signaled. If the referent database has been deleted, err_database_not_found is signaled.

os_reference_protected

Instances of the class **os_reference_protected** can be used as substitutes for cross-database and cross-transaction pointers. References are valid under a wider array of circumstances than are pointers to persistent storage.

A pointer to persistent storage assigned to transient memory is valid only until the end of the outermost transaction in which the assignment occurred, unless **objectstore::retain_persistent_addresses()** is used. In addition, a pointer to storage in one database assigned to storage in another database is valid only until the end of the outermost transaction in which the assignment occurred, unless **os_database::allow_external_pointers()** or **os_segment::allow_external_pointers()** is used.

os_reference_protecteds, in contrast, are always valid across transaction boundaries, as well as across databases.

Once the object referred to by an **os_reference_protected** is deleted, use of the **os_reference_protected** causes an err_reference_not_found exception to be signaled. If the referent database has been deleted, err_database_not_found is signaled.

You can create a reference to serve as substitute for a pointer by initializing a variable of type **os_reference_protected** with the pointer, or by assigning the pointer to a variable of type **os_reference_protected** (implicitly invoking the conversion constructor **os_reference_protected::os_reference_protected(void*)**). This pointer must not point to transient memory. In general, a pointer can be used anywhere an **os_reference_protected** is expected, and the conversion constructor will be invoked.

```
part *a_part = ... ;
os_reference_protected part_ref = a_part;
```

When an **os_reference_protected** is cast to pointer-to-referent-type (that is, pointer to the type of object referred to by the reference), **os_reference_protected::operator void*()** is implicitly invoked, returning a valid pointer to the object referred to by the **os_reference_protected**.

```
printf("%d\n", (part*)(part_ref)->part_id);
```

Each instance of this class stores a relative pathname to identify the referent database. The pathname is relative to the lowest common directory in the pathnames of the referent database and the database containing the reference. For example, if a reference stored in **/A/B/C/db1** refers to data in **/A/B/D/db2**, the lowest common directory is **A/B**, so the relative pathname **../../D/db2** is used.

This means that if you copy a database containing a reference, the reference in the copy and the reference in the original might refer to different databases. To change the database a reference refers to, you can use the ObjectStore utility **oschangedbref**. See *ObjectStore Management*.

Using memcpy() with persistent os_reference_ protecteds

You can use the C++ **memcpy()** function to copy a persistent **os_ reference_protected** only if the target object is in the same segment as the source object. This is because all persistent objects of the type **os_reference_protected** use **os_segment::of(this)** for reference resolution processing and the resoultion will be incorrect if the **os_reference** has been copied to a different segment.

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.

All ObjectStore programs must include the header file **<ostore/ostore.hh>**.

os_reference_protected::deleted()

os_boolean deleted() const;

Returns **1** (true) if the object to which the specified reference refers has been deleted; returns **0** (false) otherwise.

os_reference_protected::dump()

char* dump(const char* db_str) const;

Returns a heap-allocated string for the reference that the method was called. However, unlike the string returned by the **char* os_ Reference_protected::dump(void)** method, the returned string does not contain an absolute database pathname. The returned string is intended to be used as the **dump_str** parameter of an **os_ Reference_protected** load method of the form **load(const char***

dump_str, os_database* db). It is the responsibility of the caller of **load** to ensure that the **db** parameter passed to the load method is the same as the database of the dumped reference. It is the user's responsibility to delete the returned string when finished using the string.

This operation is useful in those applications in which you do not want the overhead of storing the absolute database path in the dumped strings.

os_reference_protected::forget()

void forget();

Frees the memory in the underlying table used to associate the specified reference with its referent. Subsequent use of the **os_reference_protected** will result in a run-time error.

os_reference_protected::get_database()

os_database *get_database() const;

Returns a pointer to the database containing the object referred to by the specified reference.

os_reference_protected::get_database_key();

char* get_database_key(const char* dump_str);

Returns a heap-allocated string containing the **database_key** component of the string **dump_str**. **dump_str** must have been generated using the **dump** operation. Otherwise, the exception err_reference_syntax is raised. It is the user's responsibility to delete the returned string when finished using the string.

os_reference_protected::get_open_database()

os_database *get_open_database() const;

Returns a pointer to the database containing the object referred to by the specified **os_reference_protected**. Opens the database.

os_reference_protected::get_os_typespec()

static os_typespec *get_os_typespec();

Returns an **os_typespec*** for the class **os_reference_protected**.

os_reference_protected::hash()

os_unsigned_int32 hash() const;

Returns an integer suitable for use as a hash table key. The value returned is always the same for an **os_reference_protected** to a given referent.

os_reference_protected::load()

void load(const char* dump_str, const os_database* db);

The **dump_str** parameter is assumed to be the result of a call to a compatible **os_reference dump** method. It is the responsibility of the caller of **load** to ensure that the **db** parameter passed to the load method is the same as the database of the originally dumped reference. The loaded reference refers to the same object as the **os_reference** used to dump the string as long as the **db** parameter is the same as the database of the dumped reference. The exception err_reference_syntax is raised if the **dump_str** is not in the expected format or if the **dump_str** was dumped from a nonprotected reference.

os_reference_protected::operator void*()

operator void*() const;

Returns the valid pointer for which the specified reference is a substitute. If the referent has been deleted, err_reference_not_found is signaled. If the referent database has been deleted, err_database_not_found is signaled.

os_reference_protected::operator =()

os_reference_protected &operator=(const os_reference_protected&);

Establishes the referent of the right operand as the referent of the left operand.

os_reference_protected &operator=(const void*);

Establishes the object pointed to by the right operand as the referent of the left operand.

os_reference_protected::operator ==()

os_boolean operator ==(os_reference_protected const&) const;

Returns **1** if the arguments have the same referent; returns **0** otherwise.

os_reference_protected::operator !()

os_boolean operator !() const;

Returns nonzero if the reference has no current referent.

os_reference_protected::operator !=()

os_boolean operator !=(os_reference_protected const&) const;

Returns **1** if the arguments have different referents; returns **0** otherwise.

os_reference_protected::operator <()

os_boolean operator <(os_reference_protected const&) const;

If the first and second arguments refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument precedes the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

os_reference_protected::operator >()

os_boolean operator >(os_reference_protected const&) const;

If the first and second arguments refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument follows the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

os_reference_protected::operator >=()

os_boolean operator >=(os_reference_protected const&) const;

If the first and second arguments refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument follows or is the same as the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

os_reference_protected::operator <=()

os_boolean operator <=(os_reference_protected const&) const;

If the first and second arguments refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument precedes or is the same as the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

os_reference_protected::os_reference_protected()

os_reference_protected(const void*);

Constructs an **os_reference_protected** to substitute for the specified **void***. If the **void*** points to transient memory, err_reference_to_transient is signaled. **0** is a legal argument.

os_reference_protected::resolve()

void* resolve();

Returns the **void*** for which the specified **os_reference_protected** is a substitute. If the referent has been deleted, err_reference_not_found is signaled. If the referent database has been deleted, err_database_not_found is signaled.

os_Reference_protected_local

Instances of the class **os_Reference_protected_local** can be used as substitutes for cross-database and cross-transaction pointers. References are valid under a wider array of circumstances than are pointers to persistent storage.

A pointer to persistent storage assigned to transient memory is valid only until the end of the outermost transaction in which the assignment occurred, unless **objectstore::retain_persistent_addresses()** is used. In addition, a pointer to storage in one database assigned to storage in another database is valid only until the end of the outermost transaction in which the assignment occurred, unless **os_database::allow_external_pointers()** or **os_segment::allow_external_pointers()** is used.

os_Reference_protected_locals, in contrast, are always valid across transaction boundaries, as well as across databases.

Once the object referred to by an **os_Reference_protected_local** is deleted, use of the **os_Reference_protected_local** will cause err_reference_not_found to be signaled. If the referent database has been deleted, err_database_not_found is signaled.

The class **os_Reference_protected_local** is *parameterized*, with a parameter for indicating the type of the object referred to by a reference. This means that when specifying **os_Reference_protected_local** as a function's formal parameter, or as the type of a variable or data member, you must specify the parameter — the reference's *referent type*. You do this by appending to **os_Reference_protected_local** the name of the referent type enclosed in angle brackets (< >):

os_Reference_protected_local<*referent-type-name*>

The referent type must be a class. For protected local references to built-in types, such as **int** and **char**, see **os_reference_protected_local** on page 242.

The referent type parameter, **T**, occurs in the signatures of some of the functions described below. The parameter is used by the compiler to detect type errors.

You can create a reference to serve as substitute for a pointer of type **T*** by initializing a variable of type **os_Reference_protected_**

local<T> with a **T***, or by assigning a **T*** to a variable of type **os_Reference_protected_local<T>** (implicitly invoking the conversion constructor **os_Reference_protected_local::os_Reference_protected_local(T*)**). This pointer must not point to transient memory.

```
part *a_part = ... ;
os_Reference_protected_local<part> part_ref = a_part;
```

When the member function **resolve()** is applied to an **os_Reference_protected_local**, with a pointer to the referent database as argument, a valid pointer to the referent object is returned.

```
printf("%d\n", part_ref.resolve(db1)->part_id);
```

In some cases involving multiple inheritance, comparing two references has a different result from comparing the corresponding pointers. For example, for **==** comparisons, if the referent type of one operand is a nonleftmost base class of the referent type of the other operand, the result is always 1. This is because comparing references never results in the pointer adjustment described in Section 10.3c of the *C++ Annotated Reference Manual*.

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.

All ObjectStore programs must include the header file **<ostore/ostore.hh>**.

os_Reference_protected_local::deleted()

os_boolean deleted(os_database *db) const;

Returns **1** (true) if the object to which the specified reference refers has been deleted from the specified database; **0** (false) otherwise.

os_Reference_protected_local::dump()

char *dump(const char *database_name) const;

Returns a heap-allocated text string representing the specified reference. When this string is passed to **os_Reference_protected_local::load()**, the result is a reference to the same object referred to by the dumped reference. It is the user’s responsibility to delete the returned string.

os_Reference_protected_local::get_database_key()

char* get_database_key(const char* dump_str);

Returns a heap-allocated string containing the **database_key** component of the string **dump_str**. **dump_str** must have been generated using the **dump** operation. Otherwise, the exception err_reference_syntax is raised. It is the user's responsibility to delete the returned string when finished using the string.

os_Reference_protected_local::forget()

void forget(os_database *db);

Frees the memory in the underlying table used to associate the specified **os_Reference_protected_local** with its referent in the specified database. Subsequent use of the **os_Reference_protected_local** will result in a run-time error.

os_Reference_protected_local::get_os_typespec()

static os_typespec *get_os_typespec();

Returns an **os_typespec*** for the class **os_Reference_protected_local**.

os_Reference_protected_local::hash()

os_unsigned_int32 hash() const;

Returns an integer suitable for use as a hash table key. The value returned is always the same for an **os_Reference_protected_local** to a given referent.

os_Reference_protected_local::load()

void load(const char*);

If the specified **char*** points to a string generated from a reference with **os_Reference_protected_local::dump()**, calling this function makes the specified reference refer to the same object referred to by the reference used to generate the string.

os_Reference_protected_local::operator =()

```
os_Reference_protected_local<T> &operator=(
    const os_Reference_protected_local<T>&
);
```

Establishes the referent of the right operand as the referent of the left operand.

os_Reference_protected_local<T> &operator=(const T*);

Establishes the object pointed to by the right operand as the referent of the left operand.

os_Reference_protected_local::operator ==()

os_boolean operator ==(os_Reference_protected_local const&) const;

Returns **1** if the arguments have the same referent; returns **0** otherwise.

os_Reference_protected_local::operator !()

os_boolean operator !() const;

Returns nonzero if the reference has no current referent.

os_Reference_protected_local::operator !=()

os_boolean operator !=(os_Reference_protected_local const&) const;

Returns **1** if the arguments have different referents; returns **0** otherwise.

os_Reference_protected_local::operator <()

os_boolean operator <(os_Reference_protected_local const&) const;

If the first argument and second argument refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument precedes the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

os_Reference_protected_local::operator >()

os_boolean operator >(os_Reference_protected_local const&) const;

If the first argument and second argument refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument follows the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

os_Reference_protected_local::operator >=()

os_boolean operator >=(os_Reference_protected_local const&) const;

If the first argument and second argument refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument follows or is the same as the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

os_Reference_protected_local::operator <=()

os_boolean operator <=(os_Reference_protected_local const&) const;

If the first argument and second argument refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument precedes or is the same as the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

os_Reference_protected_local::os_Reference_protected_local()

os_Reference_protected_local(T*);

Constructs a reference to substitute for the specified **T***. If the **T*** points to transient memory, err_reference_to_transient is signaled. **0** is a legal argument.

os_Reference_protected_local::resolve()

T* resolve(const os_database*) const;

Returns the **T*** for which the specified **os_Reference_protected_local** is a substitute. The database containing the storage pointed to by the **T*** must be specified. If the referent has been deleted, err_reference_not_found is signaled. If the referent database has been deleted, err_database_not_found is signaled.

os_reference_protected_local

Instances of the class **os_reference_protected_local** can be used as substitutes for cross-database and cross-transaction pointers. References are valid under a wider array of circumstances than are pointers to persistent storage.

A pointer to persistent storage assigned to transient memory is valid only until the end of the outermost transaction in which the assignment occurred, unless **objectstore::retain_persistent_addresses()** is used. In addition, a pointer to storage in one database assigned to storage in another database is valid only until the end of the outermost transaction in which the assignment occurred, unless **os_database::allow_external_pointers()** or **os_segment::allow_external_pointers()** is used.

os_reference_protected_locals, in contrast, are always valid across transaction boundaries, as well as across databases.

Once the object referred to by an **os_reference_protected_local** is deleted, use of the **os_reference_protected_local** will cause an err_reference_not_found exception to be signaled. If the referent database has been deleted, err_database_not_found is signaled.

You can create a reference to serve as substitute for a pointer by initializing a variable of type **os_reference_protected_local** with the pointer, or by assigning the pointer to a variable of type **os_reference_protected_local** (implicitly invoking the conversion constructor **os_reference_protected_local::os_reference_protected_local(void*)**). This pointer must not point to transient memory. In general, a pointer can be used anywhere an **os_reference_protected_local** is expected, and the conversion constructor will be invoked.

```
part *a_part = ... ;
os_reference_protected_local part_ref = a_part;
```

When the member function **resolve()** is applied to an **os_reference_protected_local**, with a pointer to the referent database as argument, a valid pointer to the referent object is returned.

```
printf("%d\n", part_ref.resolve(db1)->part_id);
```

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.

All ObjectStore programs must include the header file **<ostore/ostore.hh>**.

os_reference_protected_local::deleted()

os_boolean deleted(os_database *db) const;

Returns **1** (true) if the object to which the specified reference refers has been deleted from the specified database; **0** (false) otherwise.

os_reference_protected_local::dump()

char *dump(const char *database_name) const;

Returns a heap-allocated text string representing the specified reference. When this string is passed to **os_reference_protected_ local::load()**, the result is a reference to the same object referred to by the dumped reference. It is the user's responsibility to delete the returned string.

os_reference_protected_local::forget()

void forget(os_database *db);

Frees the memory in the underlying table used to associate the specified **os_reference_protected_local** with its referent in the specified database. Subsequent use of the **os_reference_ protected_local** will result in a run-time error.

os_reference_protected_local::get_database_key()

char* get_database_key(const char* dump_str);

Returns a heap-allocated string containing the **database_key** component of the string **dump_str**. **dump_str** must have been generated using the **dump** operation. Otherwise, the exception err_reference_syntax is raised. It is the user's responsibility to delete the returned string when finished using the string.

os_reference_protected_local::get_os_typespec()

static os_typespec *get_os_typespec();

Returns an **os_typespec*** for the class **os_reference_protected_ local**.

os_reference_protected_local::hash()

os_unsigned_int32 hash() const;

Returns an integer suitable for use as a hash table key. The value returned is always the same for an **os_reference_protected_local** to a given referent.

os_reference_protected_local::load()

void load(const char*);

If the specified **char*** points to a string generated from a reference with **os_reference_protected_local::dump()**, calling this function makes the specified reference refer to the same object referred to by the reference used to generate the string.

os_reference_protected_local::operator =()

```
os_reference_protected_local &operator=(
    const os_reference_protected_local&
);
```

Establishes the referent of the right operand as the referent of the left operand.

os_reference_protected_local &operator=(const void*);

Establishes the object pointed to by the right operand as the referent of the left operand.

os_reference_protected_local::operator ==()

os_boolean operator ==(os_reference_protected_local const&) const;

Returns **1** if the arguments have the same referent; returns **0** otherwise.

os_reference_protected_local::operator !()

os_boolean operator !() const;

Returns nonzero if the reference has no current referent.

os_reference_protected_local::operator !=()

```
os_boolean operator !=(
    os_reference_protected_local const&
) const;
```

Returns **1** if the arguments have different referents; returns **0** otherwise.

os_reference_protected_local::operator <()

os_boolean operator <(os_reference_protected_local const&) const;

If the first and second arguments refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument precedes the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

os_reference_protected_local::operator >()

os_boolean operator >(os_reference_protected_local const&) const;

If the first and second arguments refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument follows the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

os_reference_protected_local::operator >=()

os_boolean operator >=(os_reference_protected_local const&) const;

If the first and second arguments refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument follows or is the same as the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

os_reference_protected_local::operator <=()

os_boolean operator <=(os_reference_protected_local const&) const;

If the first and second arguments refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument precedes or is the same as the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

os_reference_protected_local::os_reference_protected_local()

os_reference_protected_local(const void *);

Constructs a reference to substitute for the specified **void***. If the **void*** points to transient memory, err_reference_to_transient is signaled. **0** is a legal argument.

os_reference_protected_local::resolve()

void* resolve(const os_database*);

Returns the **void*** for which the specified **os_reference_protected_local** is a substitute. The database containing the storage pointed to by the **void*** must be specified. If the referent has been deleted, err_reference_not_found is signaled.

os_Reference_this_DB

Instances of the class **os_Reference_this_DB** can be used as substitutes for intradatabase and cross-transaction pointers. The reference and the referent must be in the same database.

Once the object referred to by a reference is deleted, use of the reference accesses arbitrary data and might cause a segmentation violation.

The class **os_Reference_this_DB** is *parameterized*, with a parameter for indicating the type of the object referred to by a reference. This means that when specifying **os_Reference_this_DB** as a function's formal parameter, or as the type of a variable or data member, you must specify the parameter — the reference's *referent type*. You do this by appending to **os_Reference_this_DB** the name of the referent type enclosed in angle brackets (< >):

os_Reference_this_DB<*referent-type-name*>

The referent type must be a class. For references to built-in types, such as **int** and **char**, see **os_reference_this_DB** on page 252.

The referent type parameter, **T**, occurs in the signatures of some of the functions described below. The parameter is used by the compiler to detect type errors.

You can create a reference to serve as substitute for a pointer of type **T*** by initializing a variable of type **os_Reference_this_DB<T>** with a **T***, or by assigning a **T*** to a variable of type **os_Reference_this_DB<T>** (implicitly invoking the conversion constructor **os_Reference_this_DB::os_Reference_this_DB(T*)**).

```
part *a_part = ... ;
os_Reference_this_DB<part> part_ref = a_part;
```

When an **os_Reference_this_DB<T*>** is used where a **T*** is expected, **os_Reference_this_DB::operator ->()** or **os_Reference_this_DB::operator T*()** is implicitly invoked, returning a valid pointer to the object referred to by the **os_Reference_this_DB**.

```
printf("%d\n", part_ref->part_id);
```

Not all C++ operators have special reference class overloadings. References do not behave like pointers in the context of **[]** and **++**, for example.

In some cases involving multiple inheritance, comparing two references has a different result from comparing the corresponding pointers. For example, for **==** comparisons, if the referent type of one operand is a nonleftmost base class of the referent type of the other operand, the result is always 1. This is because comparing references never results in the pointer adjustment described in Section 10.3c of the *C++ Annotated Reference Manual*.

Using memcpy() with persistent os_Reference_this_DBs

You can use the C++ **memcpy()** function to copy a persistent **os_Reference_this_DB** only if the target object is in the same segment as the source object. This is because all persistent objects of this type use **os_segment::of(this)** for reference resolution processing and the resolution will be incorrect if the **os_Reference_this_DB** has been copied to a different segment.

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.

All ObjectStore programs must include the header file **<ostore/ostore.hh>**.

os_Reference_this_DB::dump()

char* dump(const char* db_str) const;

Returns a heap-allocated string for the reference that the method was called. However, unlike the string returned by the **char* os_Reference_this_DB::dump(void)** method, the returned string does not contain an absolute database pathname. The returned string is intended for use as the **dump_str** parameter of an **os_Reference_this_DB** load method of the form **load(const char* dump_str, os_database* db)**. It is the responsibility of the caller of **load** to ensure that the **db** parameter passed to the load method is the same as the database of the dumped reference. It is the user’s responsibility to delete the returned string when finished using the string.

This operation is useful in those applications in which you do not want the overhead of storing the absolute database path in the dumped strings.

os_Reference_this_DB::get_database()

os_database *get_database() const;

Returns a pointer to the database containing the object referred to by the specified reference.

os_Reference_this_DB:get_database_key();

char* get_database_key(const char* dump_str);

Returns a heap-allocated string containing the **database_key** component of the string **dump_str**. **dump_str** must have been generated using the **dump** operation. Otherwise, the exception err_reference_syntax is raised. It is the user's responsibility to delete the returned string when finished using the string.

os_Reference_this_DB::get_os_typespec()

static os_typespec *get_os_typespec();

Returns an **os_typespec*** for the class **os_Reference_this_DB**.

os_Reference_this_DB::hash()

os_unsigned_int32 hash() const;

Returns an integer suitable for use as a hash table key. The value returned is always the same for a reference to a given referent.

os_Reference_this_DB::load()

void load(const char* dump_str, const os_database* db);

The **dump_str** parameter is assumed to be the result of a call to a compatible **os_Reference dump** method. It is the responsibility of the caller of **load** to ensure that the **db** parameter passed to the load method is the same as the database of the originally dumped reference. The loaded reference refers to the same object as the **os_Reference** used to dump the string as long as the **db** parameter is the same as the database of the dumped reference. The exception err_reference_syntax is raised if the **dump_str** is not in the expected format.

os_Reference_this_DB::operator T*()

operator T*() const;

Returns the valid **T*** for which the specified reference is a substitute.

os_Reference_this_DB::operator –>()

T* operator –>() const;

Returns the valid **T*** for which the specified reference is a substitute.

os_Reference_this_DB::operator =()

os_Reference_this_DB<T> &operator=(
const os_Reference_this_DB<T>&
);

Establishes the referent of the right operand as the referent of the left operand.

os_Reference_this_DB<T> &operator=(const T*);

Establishes the object pointed to by the right operand as the referent of the left operand.

os_Reference_this_DB::operator ==()

os_boolean operator ==(os_Reference_this_DB const&) const;

Returns **1** if the arguments have the same referent; returns **0** otherwise.

os_Reference_this_DB::operator !=()

os_boolean operator !=(os_Reference_this_DB const&) const;

Returns **1** if the arguments have different referents; returns **0** otherwise.

os_Reference_this_DB::operator <()

os_boolean operator <(os_Reference_this_DB const&) const;

If the first argument and second argument refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument precedes the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

os_Reference_this_DB:operator >()

os_boolean operator >(os_Reference_this_DB const&) const;

If the first argument and second argument refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument follows the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

os_Reference_this_DB:operator >=()

os_boolean operator >=(os_Reference_this_DB const&) const;

If the first argument and second argument refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument follows or is the same as the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

os_Reference_this_DB:operator <=()

os_boolean operator <=(os_Reference_this_DB const&) const;

If the first argument and second argument refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument precedes or is the same as the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

os_Reference_this_DB::os_Reference_this_DB()

os_Reference_this_DB(T*);

Constructs a reference to substitute for the specified **T***.

os_Reference_this_DB::resolve()

T *resolve() const;

Returns the valid **T*** for which the specified reference is a substitute.

os_reference_this_DB

Instances of the class **os_reference_this_DB** can be used as substitutes for intradatabase and cross-transaction pointers. The reference and the referent must be in the same database.

Once the object referred to by a reference is deleted, use of the reference accesses arbitrary data and might cause a segmentation violation.

You can create a reference to serve as substitute for a pointer by initializing a variable of type **os_reference_this_DB** with the pointer, or by assigning the pointer to a variable of type **os_ reference_this_DB** (implicitly invoking the conversion constructor **os_reference_this_DB::os_reference_this_DB(void*)**). In general, a pointer can be used anywhere an **os_reference_this_DB** is expected, and the conversion constructor will be invoked**.**

```
part *a_part = ... ;
os_reference_this_DB part_ref = a_part;
```

When an **os_reference_this_DB** is cast to pointer-to-referent-type (that is, pointer to the type of object referred to by the reference), **os_reference_this_DB::operator void*()** is implicitly invoked, returning a valid pointer to the object referred to by the **os_ reference_this_DB**.

```
printf("%d\n", (part*)(part_ref)->part_id);
```

Performing the member function **resolve()** on an **os_reference_ this_DB** returns a valid pointer to the object referred to by the **os_ reference_this_DB**.

```
printf("%d\n", part_ref.>resolve()->part_id);
```

Using memcpy() with persistent os_reference_this_DBs

You can use the C++ **memcpy()** function to copy a persistent **os_ reference_this_DB** only if the target object is in the same segment as the source object. This is because persistent objects of the type **os_reference_this_DB** use **os_segment::of(this)** for reference resolution processing and the resolution will be incorrect if the **os_reference_this_DB** has been copied to a different segment.

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.

All ObjectStore programs must include the header file **<ostore\ostore.hh>**.

os_reference_this_DB::dump()

char* dump(const char* db_str) const;

Returns a heap-allocated string for the reference that the method was called. However, unlike the string returned by the **char* os_ reference_this_DB::dump(void)** method, the returned string does not contain an absolute database pathname. The returned string is intended for use as the **dump_str** parameter of an **os_reference_ this_DB** load method of the form **load(const char* dump_str, os_ database* db)**. It is the responsibility of the caller of **load** to ensure that the **db** parameter passed to the load method is the same as the database of the dumped reference. It is the user's responsibility to delete the returned string when finished using the string.

This operation is useful in those applications in which you do not want the overhead of storing the absolute database path in the dumped strings.

os_reference_this_DB::get_database()

os_database *get_database() const;

Returns a pointer to the database containing the object referred to by the specified reference.

os_reference_this_DB::get_database_key()

char* get_database_key(const char* dump_str);

Returns a heap-allocated string containing the **database_key** component of the string **dump_str**. **dump_str** must have been generated using the **dump** operation. Otherwise, the exception err_reference_syntax is raised. It is the user's responsibility to delete the returned string when finished using the string.

os_reference_this_DB::get_os_typespec()

static os_typespec *get_os_typespec();

Returns an **os_typespec*** for the class **os_reference_this_DB**.

os_reference_this_DB::hash()

os_unsigned_int32 hash() const;

Returns an integer suitable for use as a hash table key. The value returned is always the same for a reference to a given referent.

os_reference_this_DB::load()

void load(const char* dump_str, const os_database* db);

The **dump_str** parameter is assumed to be the result of a call to a compatible **os_reference dump** method. It is the responsibility of the caller of **load** to ensure that the **db** parameter passed to the load method is the same as the database of the originally dumped reference. The loaded reference refers to the same object as the **os_reference** used to dump the string as long as the **db** parameter is the same as the database of the dumped reference. The exception err_reference_syntax is raised if the **dump_str** is not in the expected format.

os_reference_this_DB::operator void*()

operator void*() const;

Returns the valid pointer for which the specified reference is a substitute.

os_reference_this_DB::operator =()

```
os_reference_this_DB &operator=(
    const os_reference_this_DB&
);
```

Establishes the referent of the right operand as the referent of the left operand.

os_reference_this_DB &operator=(const void*);

Establishes the object pointed to by the right operand as the referent of the left operand.

os_reference_this_DB::operator ==()

os_boolean operator ==(os_reference_this_DB const&) const;

Returns **1** if the arguments have the same referent; returns **0** otherwise.

os_reference_this_DB::operator !=()

os_boolean operator !=(os_reference_this_DB const&) const;

Returns **1** if the arguments have different referents; returns **0** otherwise.

os_reference_this_DB::operator <()

os_boolean operator <(os_reference_this_DB const&) const;

If the first argument and second argument refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument precedes the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

os_reference_this_DB::operator >()

os_boolean operator >(os_reference_this_DB const&) const;

If the first argument and second argument refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument follows the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

os_reference_this_DB::operator >=()

os_boolean operator >=(os_reference_this_DB const&) const;

If the first argument and second argument refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument follows or is the same as the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

os_reference_this_DB::operator <=()

os_boolean operator <=(os_reference_this_DB const&) const;

If the first argument and second argument refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument precedes or is the same as the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

os_reference_this_DB::os_reference_this_DB()

os_reference_this_DB(const void*);

Constructs a reference to substitute for the specified **void***.

os_reference_this_DB::resolve()

void* resolve() const;

Returns the valid **void*** for which the specified reference is a substitute.

os_Reference_transient

Instances of the class **os_Reference_transient** can be used as substitutes for cross-transaction pointers. The reference must be allocated transiently.

Once the object referred to by a reference is deleted, use of the reference accesses arbitrary data and might cause a segmentation violation.

The class **os_Reference_transient** is *parameterized*, with a parameter for indicating the type of the object referred to by a reference. This means that when specifying **os_Reference_transient** as a function's formal parameter, or as the type of a variable or data member, you must specify the parameter — the reference's *referent type*. You do this by appending to **os_Reference_transient** the name of the referent type enclosed in angle brackets (< >):

os_Reference_transient<*referent-type-name*>

The referent type must be a class. For transient references to built-in types, such as **int** and **char**, see **os_reference_transient** on page 262.

The referent type parameter, **T**, occurs in the signatures of some of the functions described below. The parameter is used by the compiler to detect type errors.

You can create a reference to serve as substitute for a pointer of type **T*** by initializing a variable of type **os_Reference_transient<T>** with a **T***, or by assigning a **T*** to a variable of type **os_Reference_transient<T>** (implicitly invoking the conversion constructor **os_Reference_transient::os_Reference_transient(T*)**).

```
part *a_part = ... ;
os_Reference_transient<part> part_ref = a_part;
```

When an **os_Reference_transient<T*>** is used where a **T*** is expected, **os_Reference_transient::operator ->()** or **os_Reference_transient::operator T*()** is implicitly invoked, returning a valid pointer to the object referred to by the **os_Reference_transient**.

```
printf("%d\n", part_ref->part_id);
```

Not all C++ operators have special reference class overloadings. References do not behave like pointers in the context of **[]** and **++**, for example.

In some cases involving multiple inheritance, comparing two references has a different result from comparing the corresponding pointers. For example, for **==** comparisons, if the referent type of one operand is a nonleftmost base class of the referent type of the other operand, the result is always 1. This is because comparing references never results in the pointer adjustment described in Section 10.3c of the *C++ Annotated Reference Manual*.

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.

All ObjectStore programs must include the header file **<ostore/ostore.hh>**.

os_Reference_transient::dump()

char* dump(const char* db_str) const;

Returns a heap-allocated string representing the specified reference. However, unlike the string returned by **char* os_ Reference_transient::dump(void)** method, the returned string does not contain an absolute database pathname. The returned string is intended to be used as the **dump_str** parameter of an **os_Reference** load method of the form **load(const char* dump_str, os_database* db)**. It is the responsibility of the caller of **load** to ensure that the **db** parameter passed to the load method is the same as the database of the dumped reference. It is the user's responsibility to delete the returned string when finished using the string.

This operation is useful in those applications in which you do not want the overhead of storing the absolute database path in the dumped strings.

os_Reference_transient::get_database_key();

char* get_database_key(const char* dump_str);

Returns a heap-allocated string containing the **database_key** component of the string **dump_str**. **dump_str** must have been generated using the **dump** operation. Otherwise, the exception err_reference_syntax is raised. It is the user's responsibility to delete the returned string.

os_Reference_transient::hash()

os_unsigned_int32 hash() const;

Returns an integer suitable for use as a hash table key. The value returned is always the same for a reference to a given referent.

os_Reference_transient::load()

void load(const char* dump_str, const os_database* db);

The **dump_str** parameter is assumed to be the result of a call to a compatible **os_Reference_transient** dump method. It is the responsibility of the caller of **load** to ensure that the **db** parameter to the load method is the same as the database of the originally dumped reference. The loaded reference will refer to the same object as the **os_Reference** used to dump the string as long as the **db** parameter is the same as the database of the dumped reference. The exception err_reference_syntax is raised if the **dump_str** is not in the expected format.

os_Reference_transient::operator T*()

operator T*() const;

Returns the valid **T*** for which the specified reference is a substitute.

os_Reference_transient::operator –>()

T* operator –>() const;

Returns the valid **T*** for which the specified reference is a substitute.

os_Reference_transient::operator =()

```
os_Reference_transient<T> &operator=(
    const os_Reference_transient<T>&
);
```

Establishes the referent of the right operand as the referent of the left operand.

os_Reference_transient<T> &operator=(const T*);

Establishes the object pointed to by the right operand as the referent of the left operand.

os_Reference_transient::operator ==()

os_boolean operator ==(os_Reference_transient const&) const;

Returns **1** if the arguments have the same referent; returns **0** otherwise.

os_Reference_transient::operator !=()

os_boolean operator !=(os_Reference_transient const&) const;

Returns **1** if the arguments have different referents; returns **0** otherwise.

os_Reference_transient::operator <()

os_boolean operator <(os_Reference_transient const&) const;

If the first argument and second argument refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument precedes the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

os_Reference_transient:operator >()

os_boolean operator >(os_Reference_transient const&) const;

If the first argument and second argument refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument follows the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

os_Reference_transient:operator >=()

os_boolean operator >=(os_Reference_transient const&) const;

If the first argument and second argument refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument follows or is the same as the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

os_Reference_transient:operator <=()

os_boolean operator <=(os_Reference_transient const&) const;

If the first argument and second argument refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument precedes or is the same as the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

os_Reference_transient::os_Reference_transient()

os_Reference_transient(T*);

Constructs a reference to substitute for the specified **T***.

os_Reference_transient::resolve()

T *resolve() const;

Returns the valid **T*** for which the specified reference is a substitute.

os_reference_transient

Instances of the class **os_reference_transient** can be used as substitutes for cross-transaction pointers. The reference must be allocated transiently.

Once the object referred to by a reference is deleted, use of the reference accesses arbitrary data and might cause a segmentation violation.

You can create a reference to serve as substitute for a pointer by initializing a variable of type **os_reference_transient** with the pointer, or by assigning the pointer to a variable of type **os_reference_transient** (implicitly invoking the conversion constructor **os_reference_transient::os_reference_transient(void*)**). In general, a pointer can be used anywhere an **os_reference_transient** is expected, and the conversion constructor will be invoked**.**

```
part *a_part = ... ;
os_reference_transient part_ref = a_part;
```

When an **os_reference_transient** is cast to pointer-to-referent-type (that is, pointer to the type of object referred to by the reference), **os_reference_transient::operator void*()** is implicitly invoked, returning a valid pointer to the object referred to by the **os_reference_transient**.

```
printf("%d\n", (part*)(part_ref)->part_id);
```

Performing the member function **resolve()** on an **os_reference_transient** returns a valid pointer to the object referred to by the **os_reference_transient**.

```
printf("%d\n", part_ref.>resolve()->part_id);
```

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.

All ObjectStore programs must include the header file **<ostore/ostore.hh>**.

os_reference_transient::dump()

```
char *dump(const char *db_str) const;
```

Returns a heap-allocated string representing the specified reference. However, unlike the string returned by **char* os_reference_transient::dump(void)** method, the returned string does not contain an absolute database pathname. The returned string is intended to be used as the **dump_str** parameter of **os_reference** load method of the form **load(const char* dump_str, os_database* db)**. It is the responsibility of the caller of **load** to ensure that the **db** parameter passed to the load method is the same as the database of the dumped reference. It is the user's responsibility to delete the returned string when finished using the string.

This operation is useful in those applications in which you do not want the overhead of storing the absolute database path in the dumped strings.

os_reference_transient::get_database_key()

char* get_database_key(const char* dump_str);

Returns the substring of **dump_str** that was the database key component of the dump string. **dump_str** must have been generated using the **dump** operation. Otherwise, the exception err_reference_syntax is raised. Note that this operation can be used with dump strings that contain the absolute pathname of the database.

os_reference_transient::hash()

os_unsigned_int32 hash() const;

Returns an integer suitable for use as a hash table key. The value returned is always the same for a reference to a given referent.

os_reference_transient::load()

void load(const char* dump_str, const os_database* db);

The **dump_str** parameter is assumed to be the result of a call to a compatible **os_reference_transient** dump method. It is the responsibility of the caller of **load** to ensure that the **db** parameter passed to the load method is the same as the database of the originally dumped reference. The loaded reference will refer to the same object as the **os_reference** used to dump the string as long as the **db** parameter is the same as the database of the dumped reference. The exception err_reference_syntax is raised if the **dump_str** is not in the expected format.

os_reference_transient::operator void*()

operator void*() const;

Returns the valid pointer for which the specified reference is a substitute.

os_reference_transient::operator =()

```
os_reference_transient &operator=(
    const os_reference_transient&
);
```

Establishes the referent of the right operand as the referent of the left operand.

os_reference_transient &operator=(const void*);

Establishes the object pointed to by the right operand as the referent of the left operand.

os_reference_transient::operator ==()

os_boolean operator ==(os_reference_transient const&) const;

Returns **1** if the arguments have the same referent; returns **0** otherwise.

os_reference_transient::operator !=()

os_boolean operator !=(os_reference_transient const&) const;

Returns **1** if the arguments have different referents; returns **0** otherwise.

os_reference_transient::operator <()

os_boolean operator <(os_reference_transient const&) const;

If the first argument and second argument refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument precedes the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

os_reference_transient::operator >()

os_boolean operator >(os_reference_transient const&) const;

If the first argument and second argument refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument follows the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

os_reference_transient::operator >=()

os_boolean operator >=(os_reference_transient const&) const;

If the first argument and second argument refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument follows or is the same as the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

os_reference_transient::operator <=()

os_boolean operator <=(os_reference_transient const&) const;

If the first argument and second argument refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument precedes or is the same as the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

os_reference_transient::os_reference_transient()

os_reference_transient(const void*);

Constructs a reference to substitute for the specified **void***.

os_reference_transient::resolve()

void* resolve() const;

Returns the valid **void*** for which the specified reference is a substitute.

os_reference_type

class os_reference_type : public os_pointer_type

This class is part of the ObjectStore metaobject protocol, which provides access to ObjectStore schemas. An instance of this class represents a C++ reference type. This class is derived from **os_pointer_type**. Performing **os_pointer_type::get_target_type()** on an **os_reference_type** results in the reference type's target type.

os_reference_type::create()

static os_reference_type &create(os_type* target);

The argument initializes the attribute **target**.

os_relationship_member_variable

class os_relationship_member_variable : public os_member_ variable

This class is part of the ObjectStore metaobject protocol, which provides access to ObjectStore schemas. An instance of this class represents a relationship (inverse) member. This class is derived from **os_member_variable**.

os_relationship_member_variable::get_related_class()

const os_class_type &get_related_class() const;

Returns the class at the target end of the relationship.

os_relationship_member_variable::get_related_member()

const os_relationship_member_variable &get_related_member() const;

Returns the inverse of the specified relationship member.

os_retain_address

The class **os_retain_address** allows an application to specify that certain address ranges be kept assigned across calls to **objectstore::release_persistent_addresses()** and top-level transactions. The interface to **os_retain_address** is similar to the interface to **os_pvar**. The **os_retain_address** constructor takes a pointer to a transient pointer that the application wants to remain assigned.

A single **os_retain_address** instance can track modifications made to a pointer. When the client releases address space, it iterates through all **os_retain_address** instances and dereferences their **ptr_to_ptr** to determine which address ranges should be retained. There are no error states: if **ptr_to_ptr** is NULL, or if it points to a pointer that does not point into persistent space, no error is signaled, and no address range is retained by the **os_retain_address** instance. Since address space is reserved in 64 KB units, the amount of address space reserved by a single **os_retain_address** is some multiple of 64 KB— usually, for objects 64 KB or smaller, it is just 64 KB.

Like **os_pvars**, **os_retain_address** inherits from **basic_undo**, and so all instances must be on the stack (correspond to automatic C++ variables). When an instance of **os_retain_address** is deleted, it is removed from consideration by the client.

More than one instance of **os_retain_address** can refer to the same persistent address. As long as at least one instance of **os_retain_address** refers to a persistent address when the client releases addresses, that persistent address is retained.

The retaining function member returns the persistent address pointed to by the **ptr_to_ptr** data member, or NULL if the **ptr_to_ptr** is NULL. Calling the release function member is equivalent to calling **set_retain(**NULL**)**.

Use of Pvars

Instances of **os_pvar** are treated specially by the address release operation when called within a transaction. Any such **os_pvars** that are “active” when address space is released act like instances of **os_retain_address** — the persistent address that they refer to continues to be assigned. However, unlike **os_retain_address**, active **os_pvars** do not hold address space across transaction boundaries when **objectstore::retain_persistent_addresses()** is not operating

os_retain_address::os_retain_address()

os_retain_address::os_retain_address(void **ptr_to_ptr);

os_retain_address::retaining()

void *os_retain_address::retaining() const;

os_retain_address::release()

void os_retain_address::release();

os_retain_address::set_retain()

void os_retain_address::set_retain(void **ptr_to_ptr);

The **os_retain_address::set_retain()** function member can be used to change the pointer to a transient pointer.

.

os_schema

This class is part of the ObjectStore metaobject protocol, which provides access to ObjectStore schemas. An instance of this abstract base class represents a schema. The classes **os_comp_ schema**, **os_app_schema**, and **os_database_schema** are derived from **os_schema**.

Programs using this class must include **<ostore/ostore.hh>**, followed by **<ostore/coll.hh>** (if used), followed by **<ostore/mop.hh>**.

os_schema::get_classes()

```
os_collection get_classes() const;
```

Returns a collection of the classes in the schema. Each element of the returned collection points to a **const os_class_type**.

os_schema::get_kind()

```
enum os_schema_kind {
    Compilation_schema,
    Application_schema,
    Database_schema
};
```

```
os_schema_kind get_kind () const;
```

Returns an enumerator indicating the kind of the specified schema.

os_schema::find_type()

```
const os_type *find_type(const char *typename) const;
```

Returns a pointer to the type with the specified name in the specified schema. The name can designate a class or any C++ fundamental type. All pointer types are treated identically, and result in the type for **void***'s being returned. If there is no type with the specified name, **0** is returned. For nested classes, the name must be a fully qualified name that describes the path to the nested class, for example, **outer::inner**.

os_schema::operator const os_app_schema&()

```
operator const os_app_schema&() const;
```

Provides safe conversion to **const os_app_schema&**. If the conversion is not permissible, err_mop_illegal_cast is signaled.

os_schema::operator const os_comp_schema&()

operator const os_comp_schema&() const;

Provides safe conversion to **const os_comp_schema&**. If the conversion is not permissible, err_mop_illegal_cast is signaled.

os_schema::operator const os_database_schema&()

operator const os_database_schema&() const;

Provides safe conversion to **const os_database_schema&**. If the conversion is not permissible, err_mop_illegal_cast is signaled.

os_schema::operator os_app_schema&()

operator os_app_schema&();

Provides safe conversion to **os_app_schema&**. If the conversion is not permissible, err_mop_illegal_cast is signaled.

os_schema::operator os_comp_schema&()

operator os_comp_schema&();

Provides safe conversion to **os_comp_schema&**. If the conversion is not permissible, err_mop_illegal_cast is signaled.

os_schema::operator os_database_schema&()

operator os_database_schema&();

Provides safe conversion to **os_database_schema&**. If the conversion is not permissible, err_mop_illegal_cast is signaled.

os_schema_evolution

This class provides the user interface to the ObjectStore schema evolution facility. The term *schema evolution* refers to the changes undergone by a database's schema during the course of the database's existence. It refers especially to schema changes that potentially require changing the representation of objects already stored in the database.

The schema evolution process has two phases:

- *Schema modification*: modification of the schema information associated with the database(s) being evolved
- *Instance migration*: modification of any existing instances of the modified classes

Instance migration itself has two phases:

- Instance initialization
- Instance transformation

Instance initialization modifies existing instances of modified classes so that their representations conform to the new class definitions. This might involve adding or deleting fields or subobjects, changing the type of a field, or deleting entire objects. This phase of migration also initializes any storage components that have been added or that have changed type.

In most cases, new fields are initialized with zeros. There is one useful exception to this, however. In the case where a field has changed type, and the old and new types are assignment compatible, the new field is initialized by assignment from the old field value. See **os_schema_evolution::evolve()** on page 278 for the initialization rules.

During the initialization phase, the address of an instance being migrated generally changes. The reason for this is that migration actually consists of making a copy of the old, unmigrated instance, and then modifying this copy. The copy and the old instance will be in the same segment, but their offsets within the segment will be different.

Because of this, the schema evolution facility automatically modifies all pointers to the instance so that they point to the new, modified instance. This is done for *all* pointers in the databases being evolved, including pointers contained in instances of unmodified classes, cross-database pointers, and pointers to subobjects of migrated instances.

During this process of adjusting pointers to modified instances, ObjectStore might detect various kinds of illegal pointers (see **os_ schema_evolution::evolve()** on page 278). For example, it might detect a pointer to the value of a data member that has been removed in the new schema. Since the data member has been removed, the subobject serving as value of that data member is deleted as part of instance initialization. Any pointer to such a deleted subobject is illegal and is detected by ObjectStore.

In such a case, you can provide a special handler function (see **os_ schema_evolution::set_illegal_pointer_handler()** on page 284) to process the illegal pointer (for example, by changing it to null or simply reporting its presence). Each time an illegal pointer is detected, the handler function is executed on the pointer, and then schema evolution is resumed. If you do not provide a handler function, an exception is signaled when an illegal pointer is encountered.

C++ references are treated as a kind of pointer. References to migrated instances are adjusted just as described above. And illegal references are detected and can be handled as described.

In addition, as with pointers, ObjectStore references to migrated instances are adjusted to refer to the new instance rather than the old. You are given an option concerning local references. Recall that to resolve a local reference you must specify the database containing the referent. If you want, you can direct ObjectStore to resolve each local reference using the database in which the reference itself resides. If you do not use this option, local references will not be adjusted during instance initialization (but you can provide a transformer function so that they are adjusted during the *instance transformation* phase — see below).

As with pointers, you can supply handler functions for illegal references. If you do not supply an illegal reference handler, evolution continues uninterrupted when an illegal reference is encountered. The reference is left unmodified and no exception is signaled.

Just as some pointers and references become obsolete after schema evolution, so do some indexes and persistently stored queries. For example, the selection criterion of a query or the path of an index might make reference to a removed data member. ObjectStore detects all such queries and indexes. In the case of an obsolete query, ObjectStore internally marks the query so that subsequent attempts to use it cause a run-time error.

As with illegal pointers, you can handle obsolete queries or dropped indexes by providing a special handler function for each purpose (see **os_schema_evolution::set_obsolete_query_handler()** on page 286 and **os_schema_evolution::set_obsolete_index_handler()** on page 286). An obsolete index handler, for example, might create a new index using a path that is legal under the new schema. If you do not supply handlers, ObjectStore signals an exception when an obsolete query or index is encountered.

For some schema changes, the instance initialization phase is all that is needed. But in other cases, further modification of class instances or associated data structures is required to complete the schema evolution. This further modification is generally application dependent, so ObjectStore allows you to define your own functions, *transformer functions*, to perform the task (see **os_schema_evolution::augment_post_evol_transformers()** on page 277).

You associate exactly one transformer with each class whose instances you want to be transformed. During the transformation phase of instance migration, the schema evolution facility invokes each transformer function on each instance of the function's associated class, including instances that are subobjects of other objects.

Transformer functions are particularly useful when you want to set the value of some field of a migrated instance based on the values of some field or fields of the corresponding old instance. For this purpose, the evolution facility provides functions that allow you to access the old instance corresponding to a given new instance (see **os_schema_evolution::get_unevolved_object()** on page 284, **os_typed_pointer_void::get_type()** on page 357, **os_class_type::find_member_variable()** on page 59, and **::os_fetch()** on page 370).

You can also use a transformer function to adjust local references. A transformer, associated with the class **os_reference_local** or **os_ reference_protected_local**, could perform the adjustment by retrieving the new version of each local reference's referent (see **os_schema_evolution::get_evolved_object()** on page 283), and assigning it to the reference.

In addition, transformers are useful for updating data structures that depend on the addresses of migrated instances. A hash table, for example, that hashes on addresses should be rebuilt using a transformer. Note that you do *not* need to rebuild a data structure if the position of an entry in the structure does not depend on the address of an object pointed to by the entry, but depends instead, for example, on the value of some field of the object pointed to. Such data structures will still be correct after the instance initialization phase.

Once the transformation phase is complete, all the old, unmigrated instances are deleted. (If the old instances of a given class are not needed for the transformation phase, you can direct ObjectStore to delete them during the initialization phase — see **os_schema_evolution::augment_classes_to_be_recycled()** on page 276.)

The schema evolution facility allows for one special form of instance migration, which allows you to *reclassify* instances of a given class as instances of a class derived from the given class. This form of migration is special because it is not, strictly speaking, a case of modifying instances to conform to a new class definition — you could even reclassify instances without changing the schema at all. However, instance reclassification is typically desirable when new subclasses are added to a schema. Instances of the base class can be given a more specialized representation by being classified as instances of one of the derived classes.

Reclassification occurs during the initialization phase. You specify how instances of a given base class are to be reclassified by associating a *reclassification function* with the base class (see **os_ schema_evolution::augment_subtype_selectors()**). This function takes an instance of the base class as argument, and returns the name of the instance's new class, if it is to be reclassified.

Reclassified instances can then be transformed during the transformation phase, as with any migrated instances. A reclassified instance will be transformed by the transformer function associated with its new class, a class derived from its original class.

To help you get an overall picture of the operations involved in instance initialization for a particular evolution, the schema evolution facility allows you to obtain a *task list* describing the process. The task list consists of function definitions indicating how the instances of each class will be initialized. You can generate this list without actually invoking evolution, so you can verify your expectations concerning a particular schema change before migrating the data (see **os_schema_evolution::task_list()** on page 287).

To perform schema evolution, you make and execute an application that invokes the static member function **os_schema_ evolution::evolve()**. The function must be called outside the dynamic scope of a transaction.

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.

Programs using this class must include **<ostore/ostore.hh>,** followed by **<ostore/coll.hh>** (if used), followed by **<ostore/schmevol.hh>**.

os_schema_evolution::augment_classes_to_be_recycled()

```
static void augment_classes_to_be_recycled(
    const char *class_name
);
```

Adds the class with the specified name to the set of classes whose old, unevolved instances are to be deleted during the instance initialization phase of schema evolution. Applies to evolutions initiated in the current process after the call to this function. The old, unevolved instances of a recycled class will not be accessible during the instance transformation phase.

```
static void augment_classes_to_be_recycled(
    const os_Collection<const char*> &class_names
);
```

Adds the classes named by the elements of the specified collection to the set of classes whose old, unevolved instances are to be deleted during the instance initialization phase of schema evolution. Applies to evolutions initiated in the current process after the call to this function. The old, unevolved instances of recycled classes will *not* be accessible during the instance transformation phase.

os_schema_evolution::augment_classes_to_be_removed()

```
static void augment_classes_to_be_removed(
    const char *class_name
);
```

Adds the class with the specified name to the set of classes to be removed from the schema during subsequent evolutions. This applies to evolutions initiated in the current process. If the indicated class is actually part of the new schema, err_se_cannot_delete_class is signaled. Note that when you remove a class, **C**, you must also remove or modify any class that mentions **C** in its definition. Otherwise err_se_cannot_delete_class is signaled.

```
static void augment_classes_to_be_removed(
    const os_collection &class_names
);
```

Adds the classes named by the elements of the specified collection to the set of classes to be removed from the schema during schema evolution. This applies to evolutions initiated during the current process after the call to this function. If the indicated class is actually part of the new schema, err_se_cannot_delete_class is signaled. Note that when you remove a class, **C**, you must also remove or modify any class that mentions **C** in its definition. Otherwise err_se_cannot_delete_class is signaled.

os_schema_evolution::augment_post_evol_transformers()

```
static void augment_post_evol_transformers(
    const os_transformer_binding&
);
```

Adds the specified transformer binding to the set of transformer bindings to be used during subsequent evolutions. This applies to evolutions initiated in the current process. A transformer binding associates a class with a function so that the function is executed on each instance of the class during the instance transformation phase of evolution — see **os_transformer_binding** on page 341.

```
static void augment_post_evol_transformers(
    const os_Collection<os_transformer_binding*>&
);
```

Adds the elements of the specified collection to the set of transformer bindings to be used during subsequent evolutions. This applies to evolutions initiated in the current process. A transformer binding associates a class with a function so that the function is executed on each instance of the class during the instance transformation phase of evolution — see **os_transformer_binding** on page 341.

os_schema_evolution::augment_subtype_selectors()

```
static void augment_subtype_selectors(
    const os_evolve_subtype_fun_binding&
);
```

Adds the specified subtype function binding to the set of subtype function bindings to be used during subsequent evolutions. This applies to evolutions initiated in the current process. A subtype function binding associates a class with a function. The function is used to reclassify instances of the class as instances of a subclass of the class. The string returned by the function for a given instance indicates the instance’s new class — see **os_evolve_subtype_fun_binding** on page 138.

```
static void augment_subtype_selectors(
    const os_Collection<os_evolve_subtype_fun_binding*>&
);
```

Adds the elements of the specified collection to the set of subtype function bindings to be used during subsequent evolutions. This applies to evolutions initiated in the current process. A subtype function binding associates a class with a function. The function is used to reclassify instances of the class as instances of a subclass of the class. The string returned by the function for a given instance indicates the instance’s new class — see **os_evolve_subtype_fun_binding** on page 138.

os_schema_evolution::evolve()

```
static void evolve (
    const char *work_db_name,
    const char *db_to_evolve
);
```

Invokes schema evolution on the database named by **db_to_evolve**. The function must be called outside the dynamic scope of a transaction. If there is no database named by **work_db_name**, one with that name is created and used as the work database. If there is a database named by **work_db_name**, it must be a work database from a prior interrupted evolution performed on the database named by **db_to_evolve**. In this case, evolution resumes from the latest consistent state prior to the last interruption.

The new schema is determined by the schema of the database being evolved together with the *modification schema*.

The modification schema is the schema for the compilation or application schema database specified in the most recent call in the current process to **os_schema_evolution::set_evolved_schema_db_name()** — see below. If there is no prior call to **set_evolved_schema_db_name()**, the modification schema is the schema for the application calling **evolve()**.

The new schema is the result of merging the schema of the database(s) being evolved with the modification schema; that is, the new schema is the union of the old schema and the modification schema minus the definitions in the old schema of classes that are also defined in the modification schema.

If there are any classes present in the old schema but not in the new schema, they must be specified prior to the call to **evolve()** using **os_schema_evolution::augment_classes_to_be_removed()**.

During the instance initialization phase, the instances of modified classes are modified to conform to the layouts imposed by the new classes.

Data members whose value type has changed are initialized by assignment with the old data member value, if old and new value types are assignment compatible. That is, ObjectStore assigns the value of the old data member to the storage associated with the new member, applying only standard conversions defined by the C++ language.

In some cases schema evolution considers types assignment compatible when C++ would not. For example, if D is derived from B, schema evolution will assign a B* to a D* if it knows that the B is also an instance of D.

If the new and old value types are not assignment compatible, then, where the new value type is a class, the new members are initialized as if by a constructor that sets each field to the appropriate representation of **0**, and where the new value type is not a class, they are initialized with the appropriate representation of **0**.

Data members added to the schema whose value type is a class are initialized as if by a constructor that sets each field to **0**. Other new data members are initialized with **0**.

Array-valued members are initialized, using the above rules, as if the i^{th} array element were a separate data member corresponding to the i^{th} element of the old data member value. If there is no i^{th} element of the old data member value (either because the old value is not an array, or because the old value is an array but does not have an i^{th} element), the new element is initialized as if by a constructor that sets each field to **0**, or with **0**.

Bit fields are evolved according to the default signed or unsigned rules of the implementation that built the evolution application. This can lead to unexpected results when an evolution application built with one default rule evolves a database originally populated by an application built by an implementation whose default rule differs. The unexpected results occur when the evolution application attempts to increase the width of a bit field.

Schema evolution cannot evolve a pointer-to-member that points to a member in a virtual base class.

When a class is modified to inherit from a base class, subobjects corresponding to the base class are initialized as if by a constructor that sets each field to **0**.

Subobjects corresponding to removed data members or base classes are deleted, as are instances of classes removed from the schema.

Changing inheritance from virtual to nonvirtual is treated as removal of the virtual base class and addition of nonvirtual base classes. Changing inheritance from nonvirtual to virtual is treated as removal of the nonvirtual base classes and addition of a virtual base class. In each case, subobjects corresponding to the added classes are initialized as if by a constructor that sets each field to **0**.

All pointers and C++ references to instances that require modification, including cross-database pointers, are adjusted to point to the new, evolved instances. All nonlocal ObjectStore references are similarly adjusted. Local references are also similarly adjusted, provided a nonzero **os_boolean** (true) is supplied as argument in the last call to **set_local_references_are_db_relative()** prior to the call to **evolve()**. Otherwise, local references are left unchanged.

Pointers, C++ references, and nonlocal ObjectStore references to deleted subobjects are detected as illegal, as are pointers and references to transient or freed memory, as well as type-mismatched pointers and references. Local ObjectStore references are also detected as illegal under the same circumstances, provided a nonzero **os_boolean** (true) is supplied as argument in the last call to **set_local_references_are_db_relative()** prior to the call to **evolve()**. Pointers of type **void*** are detected as illegal if the set of preevolution objects whose memory begins at the indicated location is changed after evolution.

Illegal pointers and references can be processed by illegal-pointer handlers supplied by the user (see **os_schema_evolution::set_illegal_pointer_handler()** on page 284).

Illegal pointers and C++ references for which there is no handler provoke the exception err_illegal_pointer or one of its child exceptions, unless **ignore_illegal_pointers** mode is on (see **os_schema_evolution::set_ignore_illegal_pointers()** on page 284).

When the selection criterion of a query or the path of an index makes reference to a removed class or data member, or makes incorrect type assumptions in light of a schema change, the query or index becomes obsolete. ObjectStore detects all obsolete queries and indexes. In the case of an obsolete query, ObjectStore internally marks the query so that subsequent attempts to use it result in the exception err_os_query_evaluation_error.

As with illegal pointers, you can handle obsolete queries and indexes by providing a special handler function for each purpose. Each obsolete index handled by such a function is automatically dropped after the function returns. If you do not supply handlers, ObjectStore signals err_schema_evolution when an obsolete query or index is detected. See **os_schema_evolution::set_obsolete_ index_handler()** on page 286 and **os_schema_evolution::set_ obsolete_query_handler()** on page 286.

Each instance of a class with an associated subtyping function is reclassified according to the class name returned by the function for the instance (see **os_schema_evolution::augment_subtype_ selectors()** on page 278).

During the instance initialization phase, unevolved instances of classes to be recycled (see **os_schema_evolution::augment_ classes_to_be_recycled()** on page 276) are deleted.

Subsequent to instance initialization, each transformer function (see **os_schema_evolution::augment_post_evol_transformers()** on page 277) is executed on each instance of its associated class. The order of execution of transformers on embedded objects follows the same pattern as constructors. When the transformer for a given class is invoked, the transformers for base classes of the given class are executed first (in declaration order), followed by the transformers for class-valued members of the given class (in declaration order), after which the transformer for the given class itself is executed.

Unevolved instances of each modified class are deleted following completion of the transformation phase.

```
static void evolve(
    const char *work_db_name,
    const os_Collection<const char*> &dbs_to_evolve
);
```

Invokes schema evolution on the databases named by the elements of **dbs_to_evolve**. The rest of the behavior for this function is as described for the previous overloading of **evolve()**, above.

```
static void evolve(
    const char *work_db_name,
    const os_Collection<const char*> &dbs_to_evolve,
    os_schema &new_schema
);
```

Invokes schema evolution on the databases named by the elements of **dbs_to_evolve**. The rest of the behavior for this function is as described for the first overloading of **evolve()**, above, except that the modification schema is specified by **new_schema**.

```
static void evolve(
    const char *work_db_name,
    const char *db_to_evolve,
    os_schema &new_schema
);
```

Invokes schema evolution on the database specified by **db_to_ evolve**. The rest of the behavior for this function is as described for the first overloading of **evolve()**, above, except that the modification schema is specified by **new_schema**.

os_schema_evolution::get_evolved_address()

static os_typed_pointer_void get_evolved_address(void*);

Returns an **os_typed_pointer_void** to the evolved object corresponding to the unevolved object pointed to by the specified **void***. The **os_typed_pointer_void** encapsulates a **void*** pointer to the evolved object. The pointer is null if the object was not evolved. Assumes that only the address of the evolved object is desired, not the object itself, and consequently does not check the validity of the object.

os_schema_evolution::get_evolved_object()

static os_typed_pointer_void get_evolved_object(void*);

Returns an **os_typed_pointer_void** to the evolved object corresponding to the unevolved object pointed to by the specified **void***. The **os_typed_pointer_void** encapsulates a **void*** pointer to the evolved object. The pointer is null if the object was not evolved. An exception is raised if this pointer is illegal (see **os_ schema_evolution::evolve()** on page 278).

os_schema_evolution::get_ignore_illegal_pointers()

static os_boolean get_ignore_illegal_pointers();

Returns nonzero if **ignore_illegal_pointers** mode is on. Returns **0** otherwise. See also **os_schema_evolution::set_ignore_illegal_ pointers()** on page 284.

os_schema_evolution::get_unevolved_address()

static os_typed_pointer_void get_unevolved_address(void*);

Returns an **os_typed_pointer_void** to the unevolved object corresponding to the evolved object pointed to by the specified **void***. Assumes that only the address of this object is desired, not the object itself, and consequently does not check the validity of the object.

os_schema_evolution::get_unevolved_object()

```
static os_typed_pointer_void get_unevolved_object(void*);
```

Returns an **os_typed_pointer_void** to the unevolved object corresponding to the evolved object pointed to by the specified **void***. An exception is raised if the unevolved object was deleted during instance initialization (see **os_schema_ evolution::augment_classes_to_be_recycled()** on page 276).

os_schema_evolution::get_work_database()

```
static os_database *get_work_database();
```

This function returns a pointer to the work database for the current evolution.

os_schema_evolution::set_evolved_schema_db_name()

```
static void set_evolved_schema_db_name(const char*);
```

Specifies the name of an application schema database used to determine the new schema in subsequent evolutions during the current process.

os_schema_evolution::set_ignore_illegal_pointers()

```
static void set_ignore_illegal_pointers(
    os_boolean)
);
```

If the argument is nonzero, turns on **ignore_illegal_pointers** mode, causing ObjectStore to ignore illegal pointers and references encountered during evolution. If the argument is **0**, turns off **ignore_illegal_pointers** mode. See also **os_schema_evolution::get_ ignore_illegal_pointers()** on page 283.

os_schema_evolution::set_illegal_pointer_handler()

```
static void set_illegal_pointer_handler(
    void (*f)(objectstore_exception_r,
        os_char_p msg, os_void_pr illegalp)
);
```

Specifies **f** as the handler function for illegal pointers. Applies to evolutions initiated in the current process after the call to this function. The function **f** must be defined by the user. The **objectstore_exception** is the exception that would have been signaled had a handler not been supplied, the **char*** points to the error message that would have been generated, and **illegalp** is a reference to the illegal pointer.

```
static void set_illegal_pointer_handler(
    void (*f)(objectstore_exception_r,
        os_char_p msg, os_canonical_ptom_r)
);
```

Specifies **f** as the handler function for illegal pointers to members. Applies to evolutions initiated in the current process after the call to this function. The function **f** must be defined by the user. The **objectstore_exception_r** is the exception that would have been signaled had a handler not been supplied, the **os_char_p** points to the error message that would have been generated, and the **os_ canonical_ptom_r** is a reference to the illegal pointer-to-member.

```
static void set_illegal_pointer_handler(
    void (*f)(objectstore_exception_r, os_char_p*msg,
        os_reference_local_r)
);
```

Specifies **f** as the handler function for illegal ObjectStore local references. Applies to evolutions initiated in the current process after the call to this function. The function **f** must be defined by the user. The **objectstore_exception_r** is the exception that would have been signaled had a handler not been supplied, the **os_char_ p*** points to the error message that would have been generated, and the **os_reference_local_r** is a C++ reference to the illegal ObjectStore reference.

```
static void set_illegal_pointer_handler(
    void (*f)(objectstore_exception_r,
        os_char_p msg, os_reference_r)
);
```

Specifies **f** as the handler function for illegal ObjectStore nonlocal references. Applies to evolutions initiated in the current process after the call to this function. The function **f** must be defined by the user. The **objectstore_exception_r** is the exception that would have been signaled had a handler not been supplied, the **os_char_ p** points to the error message that would have been generated, and the **os_reference_r** is a C++ reference to the illegal ObjectStore reference.

```
static void set_illegal_pointer_handler(
    void (*f)(objectstore_exception_r, os_char_p msg,
        os_database_root_r)
);
```

Specifies **f** as the handler function for illegal database root values. Applies to evolutions initiated in the current process after the call to this function. The function **f** must be defined by the user. The **objectstore_exception_r** is the exception that would have been signaled had a handler not been supplied, the **os_char_p*** points to the error message that would have been generated, and the **os_ database_root_r** is a C++ reference to the illegal database root.

os_schema_evolution::set_local_references_are_db_relative()

```
static void set_local_references_are_db_relative(os_boolean);
```

If a nonzero **os_boolean** (true) is supplied as argument, local references will be resolved using the database in which the reference itself resides. Otherwise local references will not be adjusted during the instance initialization phase. Applies to evolutions initiated in the current process after the call to this function.

os_schema_evolution::set_obsolete_index_handler()

```
static void set_obsolete_index_handler(
    void (*f)(const os_collection&, const char *path_string)
);
```

Specifies **f** as the handler function for obsolete indexes. Applies to evolutions initiated in the current process after the call to this function. The function **f** must be defined by the user. The **os_ collection&** is a reference to the collection whose index is obsolete, and the **char*** points to a string expressing the index's path (key).

os_schema_evolution::set_obsolete_query_handler()

```
static void set_obsolete_query_handler_handler(
    void (*f)(os_coll_query_r, os_char_const_p query_string)
);
```

Specifies **f** as the handler function for obsolete queries. Applies to evolutions initiated in the current process after the call to this function. The function **f** must be defined by the user. The **os_coll_**

query_r is a reference to the obsolete query, and the **os_char_const_p** points to a string expressing the query's selection criterion.

os_schema_evolution::set_task_list_file_name()

```
static void set_task_list_file_name(const char *file_name);
```

Specifies the file named **file_name** as the file to which a task list should be sent. Applies to task lists generated in the current process after the call to this function.

os_schema_evolution::task_list()

```
static void task_list(
    const char *work_db_name,
    const char *db_to_evolve
);
```

Generates a task list for the evolution that would have taken place had **evolve()** been called with the same arguments. The task list is sent to the file specified by the most recent call to **os_schema_evolution::set_task_list_file_name()**, or if there is no such call, to standard output. Once the task list is generated, this function exits, terminating the current process.

The task list contains a function definition for each class whose instances will be migrated. Each function has a name of the form

class-name@[1]::initializer()

where *class-name* names the function's associated class. Each *class-name*@[1]::initializer() function definition contains a statement or comment for each data member of its associated class. For a member with value type **T**, this statement or comment is either

- Assignment statement
- Call to**T@[1]::copy_initializer()**
- Call to**T@[2]::construct_initializer()**
- Call to **T@[1]::_initializer()**
- Comment indicating that the field will be **0-**initialized

An assignment statement is used when the old and new value types of the member are assignment compatible. **T@[1]::copy_initializer()** is used when the member has not been modified by the schema change, and the new value can be copied bit by bit from the old value. **T@[2]::construct_initializer()** is used when the value type has been modified and the new value type is a class. **T@[1]::initializer()** is used when the member has not been modified by the schema change, but instances of the value type of the member will be migrated. Definitions for all these functions appear in the task list.

```
static void task_list (
    const char *work_db_name,
    const os_Collection<const char*> &dbs_to_evolve
);
```

Generates a task list for the evolution that would have taken place had **evolve()** been called with the same arguments. The task list is sent to the file specified by the most recent call to **os_schema_ evolution::set_task_list_file_name()**. The contents of the task list are as described for the previous overloading of **task_list()**, above.

os_schema_handle

This transient class represents a reference or handle to a DLL (component) schema or an application schema. A schema handle is different from a schema in that the handle can exist before the schema has been loaded from its schema database. Also the handle remains valid across transactions, while a pointer to the schema is only valid for one transaction.

Include files

You must include the header file **<ostore/nreloc/schftyps.hh>**.

os_schema_handle::DLL_unloaded()

void DLL_unloaded();

If the DLL schema is loaded, marks it for unloading, otherwise does nothing and signals no error. The actual unloading, reconstruction of process type tables, and deletion of the **os_ schema_handle** occurs later (at the end of the transaction).

Unloading a DLL calls **DLL_unloaded()** from the DLL's termination function and then unloads the DLL. If the rest of the transaction tries to do anything that requires the DLL, or if the transaction is aborted and retried and the retry does not load the DLL, there is likely to be a fatal error.

It is an error to call this function with the **os_schema_handle** for an application schema. This throws the exception err_invalid_for_ application_schema.

os_schema_handle::get()

const os_app_schema& get() const;

Gets a C++ reference to an application or DLL program schema when given its **os_schema_handle.** The schema must be loaded.

If the schema is not loaded or has been unloaded, an err_schema_ not_loaded exception is thrown. This function can be called only while a transaction is in progress and the result is valid only for the duration of that transaction.

os_schema_handle::get_all()

```
static os_schema_handle** get_all(
    osbool to_load = false
);
```

With an argument of **false,** the default, this function returns a null-terminated array of pointers to **os_schema_handle** instances that are loaded and not queued to unload. This set corresponds to the current complete program schema.

With a **true** argument, this function returns a null-terminated array of pointers to the **os_schema_handle** instances that are queued to load.

The caller must deallocate the array.

os_schema_handle::get_application_schema_handle()

static os_schema_handle* get_application_schema_handle();

Returns a pointer to the **os_schema_handle** for the application schema. If the process has no application schema, this returns a pointer to a handle for a dummy schema that contains only ObjectStore's built-in types (essentially the boot schema). This function should be called only after ObjectStore has been initialized.

os_schema_handle::get_DLL_identifiers()

```
const char* const* get_DLL_identifiers(
    os_unsigned_int32& count
) const;
```

Returns an array of pointers to DLL identifiers and the number of elements in the array, given an **os_schema_handle**. The caller must not modify or deallocate the strings or the array. If the **this** argument designates an application schema, the result is null and **count** is set to zero.

os_schema_handle::get_schema_database()

os_database& get_schema_database() const;

Gets the **os_database** for the application or DLL schema database.

os_schema_handle::get_schema_database_pathname()

const char* get_schema_database_pathname() const;

Gets the application or DLL schema database pathname.

See also **objectstore::get_application_schema_pathname()**.

os_schema_handle::get_schema_info()

os_schema_info& get_schema_info() const;

Returns the **os_schema_info** that contains the DLL identifiers and schema pathname for the schema for which this is the handle.

os_schema_handle::get_status()

os_schema_handle_status get_status() const;

Returns the loaded / unloaded status of an **os_schema_handle.** The status is one of the following four symbolic constants:

- **os_schema_handle_unloaded**
- **os_schema_handle_loaded**
- **os_schema_handle_queued_to_load**
- **os_schema_handle_queued_to_unload**

The unloaded status exists only for an **os_schema_handle** that has never been loaded. Once an **os_schema_handle** leaves the **os_schema_handle_queued_to_unload** status and becomes unloaded, it is deleted.

os_schema_handle::insert_required_DLL_identifiers()

```
void insert_required_DLL_identifiers(
    os_database& db
);
```

Records all the DLL identifiers that belong to this **os_schema_handle** into the database's required DLL set. This function can be called only in an update transaction with the database open for write.

os_schema_handle::set_schema_database_pathname()

void set_schema_database_pathname (const char* path);

Sets the application or DLL schema database pathname. Must be called before the schema is loaded to be effective.

Like **objectstore::set_application_schema_pathname**, this has a 255-character limit and copies over the existing pathname.

os_schema_handle_status

This enum type is the type of the status of an **os_schema_handle**. The status can be one of **loaded**, **unloaded**, **queued_to_load**, or **queued_to_unload**.

os_schema_info

This is the common base class for **os_application_schema_info** and **os_DLL_schema_info**. It is designed for use with component schema and application schema.

Include files

You must include the header file **<ostore/nreloc/schftyps.hh>**.

os_schema_info::get()

os_schema_handle& get();

Given an **os_schema_info,** this function finds the corresponding **os_schema_handle**. Throws an err_misc exception if **this** is an **os_DLL_schema_info** and **os_DLL_schema_info::DLL_loaded()** has not been called, or **this** is an **os_application_schema_info** and the initialization that loads the application schema has not yet run.

os_schema_info::get_DLL_identifiers()

```
const char* const* get_DLL_identifiers(
    os_unsigned_int32& count
) const;
```

Returns an array of pointers to DLL identifiers and the number of elements in the array, given an **os_schema_info**. The caller must not modify or deallocate the strings or the array. If the **this** argument designates an application schema, the result is null and **count** is set to zero.

os_schema_info::get_schema_database_pathname()

const char* get_schema_database_pathname() const;

Returns the schema database pathname.

os_schema_info::set_schema_database_pathname()

void set_schema_database_pathname(const char* new_pathname);

Sets the schema database pathname. This has no useful effect if the schema has already been loaded.

Like **objectstore::set_application_schema_pathname()**, this has a 255-character limit and copies over the existing pathname.

os_schema_install_options

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.

All ObjectStore programs must include the header file **<ostore/ostore.hh>**.

os_schema_install_options::os_schema_install_options()

os_schema_install_options();

This function is the constructor for this class. The default behavior given to the class is not to copy the member function into the schema.

os_schema_install_options::set_copy_member_functions ()

void set_copy_member_functions (os_boolean_copy);

This member function is used to specify whether or not the member function information (if present) should be copied and installed into the schema during installation.

os_schema_install_options::get_copy_member_functions ()

os_boolean get_copy_member_functions ()const;

This member function returns a boolean that indicates whether the member function information (if present) is to be copied and installed into the schema during schema installation.

os_segment

ObjectStore databases are divided into *segments*. Each segment can be used as an atomic unit of transfer to and from persistent storage. Every database is created with an initial segment, the *default segment* (in which new objects are allocated by default). Databases whose schema is not stored remotely have an additional initial segment, the *schema segment* (which contains schema information used internally by ObjectStore, as well as all the database's roots). More segments can be added by the user at any time.

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.

All ObjectStore programs must include the header file **<ostore/ostore.hh>**.

os_segment::allow_external_pointers()

void allow_external_pointers();

Once invoked, cross-database pointers are allowed from the specified segment.

At any given time, a pointer from one database, db1, to another, db2, points to whichever database has a certain pathname — namely, db2's pathname at the time the pointer was stored. If db2's pathname changes (for example, as a result of performing **osmv** on db2), the pointer will no longer refer to data in db2. If some other database, db3, is given db2's original pathname (for example, as a result of performing **osmv** on db3), the pointer will refer to data in db3.

The pathname is not stored as part of the cross-database pointer (which takes the form of a regular virtual memory address), but rather as part of an **os_database_reference** stored in a table associated with the pointer.

It is illegal to rename a database so that a pointer that used to refer to another database now refers to the database in which the pointer itself resides.

os_database_references normally store a relative pathname. That is, if the source and destination databases have a common ancestor directory, the common directory is not stored as part of the pathname — only the part of the target database's pathname that is not shared with the source database's pathname is stored. The common part of the pathname is preceded by the appropriate number of **../**s to traverse the hierarchy up from the source directory to the common ancestor directory. For example, if the source and target databases are named **/sys/appl/mydb** and **/sys/lib/lib1**, respectively, the reference will store the relative pathname, **../lib/lib1**.

You can override use of relative pathnames with the functions **os_database::set_relative_directory()** on page 98 and **os_database::get_relative_directory()** on page 85 — see the entries for these functions.

Dereferencing a cross-database pointer causes the destination database, if not already open, to be opened for read / write. Thus, dereferencing such a pointer can result in error messages such as database not found.

os_segment::create_object_cluster()

os_object_cluster *create_object_cluster(os_unsigned_int32 size);

Creates an object cluster in the specified segment. The size of the new cluster in bytes is **round_up(size, page_size) – 4**. **size** must be no greater than 65536, or else err_cluster_too_big is signaled. The function returns a pointer to an instance of the class **os_object_cluster**. This instance is a transient object representing the new cluster. Note that, since it is transient, pointers to it cannot be stored in persistent memory. In particular, the return value of this function cannot be stored persistently.

os_segment::database_of()

os_database *database_of();

Returns a pointer to the database in which the specified segment resides. The transient database is returned if the transient segment is specified.

os_segment::destroy()

void destroy();

Deletes the segment for which the function is called. When a segment is destroyed, all data it contains is permanently destroyed, and pointers into the segment become invalid. Any subsequent use of the destroyed segment (such as an attempt to allocate memory within it) is an error.

os_segment::external_pointer_status()

void external_pointer_status(os_int32 *allowed, os_int32 *exist);

Sets the argument to reflect the specified segment's state with respect to cross-database pointers. **allowed** is **1** (true) if cross-database pointers are allowed in the segment, and **exist** is set to **1** (true) if the segment contains any cross-database pointers.

os_segment::get_access_control()

os_segment_access *get_access_control() const;

Returns a pointer to the segment's associated **os_segment_access**, which indicates the segment's primary group and permissions.

os_segment::get_all_object_clusters()

```
void get_all_object_clusters(
    os_int32 max_to_return,
    os_object_cluster_p *cluster_array,
    os_int32 &n_returned)
) const;
```

Provides access to all the object clusters in the specified segment. The **os_object_cluster_p*** is an array of pointers to object clusters. This array must be allocated by the user. The function **os_segment::get_n_object_clusters()** can be used to determine how large an array to allocate. **max_to_return** is specified by the user, and is the maximum number of elements the array is to have. **n_returned** refers to the actual number of segment pointers returned.

os_segment::get_application_info()

void *get_application_info() const;

Returns a pointer to the object pointed to by the pointer last passed, during the current process, to **os_segment::set_application_info()** for the specified segment. If **set_application_info()** has not been called for the specified segment during the current process, **0** is returned.

os_segment::get_check_illegal_pointers()

os_boolean get_check_illegal_pointers() const;

If the segment is in **check_illegal_pointers** mode, the function returns **1**; otherwise, it returns **0**.

os_segment::get_comment()

char *get_comment() const;

Returns a transient copy of the string associated by means of **os_segment::set_comment()** with the specified segment. If **set_comment()** has not been called for the specified segment, a zero-length string is returned. The user is responsible for deleting the returned string.

os_segment::get_database_references()

```
void get_database_references(
    os_int32 &n_refs,
    os_database_reference_p *&array
) const;
```

Allocates an array of database references, one for each database referenced by the specified segment. When the function returns, **n_refs** refers to the number of elements in the array. Note that it is the user's responsibility to deallocate the array when it is no longer needed.

os_segment::get_fetch_policy()

void get_fetch_policy(os_fetch_policy &policy, os_int32 &bytes);

Sets **policy** and **bytes** to references to an **os_fetch_policy** and integer that indicate the segment's current fetch policy. See **os_segment::set_fetch_policy()** on page 302.

os_segment::get_lock_whole_segment()

objectstore_lock_option get_lock_whole_segment();

Indicates the current locking behavior for the specified segment. **objectstore_lock_option** is an enumeration type whose enumerators are **objectstore::lock_as_used**, **objectstore::lock_segment_read**, and **objectstore::lock_segment_write**. See **os_segment::set_lock_whole_segment()** on page 303.

os_segment::get_n_object_clusters()

os_int32 get_n_object_clusters();

Returns the number of object clusters in the specified segment.

os_segment::get_null_illegal_pointers()

os_boolean get_null_illegal_pointers();

If the specified segment is in **null_illegal_pointers** mode, the function returns nonzero (that is, true); otherwise, it returns **0** (that is, false). See **os_segment::set_null_illegal_pointers()** on page 304.

os_segment::get_number()

os_unsigned_int32 get_number() const;

Returns the segment number of the specified segment. This number is suitable for passing to the **os_pathname_and_segment_number** constructor.

os_segment::get_readlock_timeout()

os_int32 get_readlock_timeout() const;

Returns the time in milliseconds for which the current process will wait to acquire a read lock on pages in the specified segment. The actual timeout is rounded up to the nearest greater number of seconds. A value of **–1** indicates that the process will wait forever if necessary.

os_segment::get_writelock_timeout()

os_int32 get_writelock_timeout() const;

Returns the time in milliseconds for which the current process will wait to acquire a write lock on pages in the specified segment. The actual timeout is rounded up to the nearest greater number of seconds. A value of **–1** indicates that the process will wait forever if necessary.

os_segment::get_transient_segment()

static os_segment *const get_transient_segment();

Returns a pointer to the *transient segment*. The transient segment can be used as argument to **new()**, to cause allocation of transient memory.

os_segment::is_deleted()

os_boolean is_deleted();

Returns a nonzero **os_boolean** (true) if the specified **os_segment** has been deleted; returns **0** (false) otherwise.

os_segment::is_empty()

os_boolean is_empty() const;

Returns a nonzero **os_boolean** (true) if the specified **os_segment** contains no nondeleted objects; returns **0** (false) otherwise.

os_segment::lock_into_cache()

void lock_into_cache();

Reduces the likelihood that the pages of a specified segment will be removed from the cache (by cache replacement). The function **os_segment::unlock_from_cache()** allows cache replacement to be performed on the segment's pages again. Note that a page's being locked or wired into the cache is independent of its locking state, in the sense of locking relevant to concurrency control; a page can be locked in the cache without being read- or write-locked.

os_segment::of()

static os_segment *of(void *location);

Returns a pointer to the segment in which the specified object resides. If the specified **void*** is **0** or points to transient memory, a pointer to the transient segment is returned (see **os_segment::get_ transient_segment()** on page 299).

os_segment::return_memory()

os_unsigned_int32 return_memory(os_boolean evict_now);

Just like **objectstore::return_memory()**, except that it acts on a specified segment rather than a specified range of addresses.

os_segment::set_access_control()

void set_access_control(const os_segment_access *new_access);

Associates the specified **os_segment_access** with the specified segment. The **os_segment_access** determines the primary group and permissions for the **os_segment**. The caller must be the owner of the database containing the specified segment.

os_segment::set_application_info()

void set_application_info(void *info);

Associates the specified object with the specified segment for the current process. The argument **info** must point to a transient object. See **os_segment::get_application_info()** on page 297.

os_segment::set_check_illegal_pointers()

void set_check_illegal_pointers(os_boolean);

At the end of each transaction, all persistently allocated data is written to database memory. Pointers written to the database that point to transient memory are *illegal* pointers. In addition, cross-database pointers from a segment that is not in **allow_external_ pointers** mode are also illegal. If you subsequently retrieve and dereference an illegal pointer, you might access arbitrary memory.

By default, ObjectStore sometimes checks for illegal pointers, but other times the checking is optimized out. However, you can instruct ObjectStore always to check for illegal pointers in a given segment or database on transaction commit.

If you pass **1** (true) to **set_check_illegal_pointers()**, **check_illegal_ pointers** mode is enabled for the specified segment. Upon commit of each transaction, for each segment in **check_illegal_pointers** mode, ObjectStore always checks each page used in the transaction. You can specify the default behavior by passing **0** (false).

The results of using this function do not remain in effect after the current process ends, and are invisible to other processes.

os_segment::set_comment()

void set_comment(char *info);

Associates a persistent copy of the specified string with the specified segment. The string must be 31 characters or fewer. The utility **ossize** prints the comment, if set, when displaying information about the segment. See *ObjectStore Management*. See also **os_segment::get_comment()** on page 298.

os_segment::set_fetch_policy()

enum os_fetch_policy { os_fetch_segment, os_fetch_page, os_ fetch_stream } ;

void set_fetch_policy(os_fetch_policy policy, os_int32 bytes);

Specifies the fetch policy for the specified segment. The policy argument should be one of the following enumerators: **os_fetch_ segment**, **os_fetch_page**, **os_fetch_stream**.

The default fetch policy is **os_fetch_page**, with a fetch quantum of 1 page (see below).

If an operation manipulates a substantial portion of a small segment, use the **os_fetch_segment** policy when performing the operation on the segment. Under this policy, ObjectStore attempts to fetch the entire segment containing the desired page in a single client / server interaction, if the segment will fit in the client cache without evicting any other data. If there is not enough space in the cache to hold the entire segment, the behavior is the same as for **os_fetch_page** with a fetch quantum specified by **bytes**.

If an operation uses a segment larger than the client cache, or does not refer to a significant portion of the segment, use the **os_fetch_ page** policy when performing the operation on the segment. This policy causes ObjectStore to fetch a specified number of bytes at a time, rounded up to the nearest positive number of pages, beginning with the page required to resolve a given object reference. **bytes** specifies the *fetch quantum*. (Note that if you specify zero bytes, this will be rounded up, and the unit of transfer will be a single page.)

The default value for the fetch quantum depends on the default page size of the machine. Appropriate values might range from 4 kilobytes to 256 kilobytes or higher, depending on the size and locality of the application data structures.

For special operations that scan sequentially through very large data structures, **os_fetch_stream** might considerably improve performance. As with **os_fetch_page**, this fetch policy lets you specify the amount of data to fetch in each client / server interaction for a particular segment. But, in addition, it specifies that a double buffering policy should be used to stream data from the segment.

This means that after the first two transfers from the segment, each transfer from the segment replaces the data cached by the *second-to-last* transfer from that segment. This way, the last two chunks of data retrieved from the segment will generally be in the client cache at the same time. And, after the first two transfers, transfers from the segment generally will not result in eviction of data from other segments. This policy also greatly reduces the internal overhead of finding pages to evict.

When you perform allocation that extends a segment whose fetch policy is **os_fetch_stream**, the double buffering described above begins when allocation reaches an offset in the segment that is aligned with the fetch quantum (that is, when the offset **mod** the fetch quantum is 0).

For all policies, if the fetch quantum exceeds the amount of available cache space (cache size minus wired pages), transfers are performed a page at a time. In general, the fetch quantum should be less than half the size of the client cache.

Note that a fetch policy established with **set_fetch_policy()** (for either a segment or a database) remains in effect only until the end of the process making the function call. Moreover, **set_fetch_ policy()** only affects transfers made by this process. Other concurrent processes can use a different fetch policy for the same segment or database.

os_segment::set_lock_whole_segment()

void set_lock_whole_segment(objectstore_lock_option);

Determines locking behavior for the specified segment. **objectstore_lock_option** is an enumeration type whose enumerators are **objectstore::lock_as_used**, **objectstore::lock_ segment_read**, and **objectstore::lock_segment_write**.

The member function **os_segment::set_fetch_policy()** controls whether or not a given segment is transferred to the client cache all at once. By default, even when a segment's pages are transferred all at once, only the page containing the referenced data is locked. You can override this default for a given segment, however, by passing **objectstore::lock_segment_read** or **objectstore::lock_segment_write** to **set_lock_whole_segment()**.

A value of **lock_segment_read** causes pages in the segment to be read-locked when cached in response to attempted access by the client. Subsequently, upgrading to read / write locks occurs on a page-by-page basis, as needed.

A value of **lock_segment_write** causes the segment's pages to be write-locked when cached in response to attempted read or write access by the client. In this case, the Server assumes from the start that write access to the entire segment is desired.

Note also that this function pertains only to the current process. The initial value of this data member for existing segments is **lock_as_used**. The initial value for segments created by the current process is the value of **os_database::set_default_lock_whole_segment()** for the database in which the segment resides.

os_segment::set_null_illegal_pointers()

void set_null_illegal_pointers(os_boolean);

By default, ObjectStore signals a run-time error when it detects an illegal pointer. If you pass **1** (true) to this function, then, for segments in **check_illegal_pointers** mode, ObjectStore changes the illegal pointer to **0** (null). You can specify the default behavior by passing **0** (false) to this function. The results of using this function do not remain in effect after the current process ends, and they are invisible to other processes. See also **os_database::set_default_null_illegal_pointers()** on page 94.

os_segment::set_readlock_timeout()

void set_readlock_timeout(os_int32);

Sets the time in milliseconds for which the current process will wait to acquire a read lock on pages in the specified segment. The actual timeout is rounded up to the nearest greater number of seconds. A value of **–1** indicates that the process should wait forever if necessary. After an attempt to acquire a read lock, if the specified time elapses without the lock's becoming available, an **os_lock_timeout** exception is signaled. If the attempt causes a deadlock, the transaction is aborted regardless of the value of the specified timeout.

os_segment::set_size()

os_unsigned_int32 set_size(os_unsigned_int32);

Increases the size of the specified segment to the specified number of bytes. If a size that is not larger than the current segment size is specified, the function has no effect.

os_segment::set_writelock_timeout()

void set_writelock_timeout(os_int32);

Sets the time in milliseconds for which the current process will wait to acquire a write lock on pages in the specified segment. The actual timeout is rounded up to the nearest greater number of seconds. A value of **–1** indicates that the process should wait forever if necessary. After an attempt to acquire a read lock, if the specified time elapses without the lock's becoming available, an **os_lock_timeout** exception is signaled. If the attempt causes a deadlock, the transaction is aborted regardless of the value of the specified timeout.

os_segment::size()

os_unsigned_int32 size();

Returns the size in bytes of the specified segment.

os_segment::unlock_from_cache()

void unlock_from_cache();

Allows cache replacement to be performed on a segment's pages after replacement has been disabled by the function **os_ segment::lock_into_cache()**. Note that a page's being locked or wired into the cache is independent of its locking state, in the sense of locking relevant to concurrency control; a page can be locked in the cache without being read- or write-locked.

os_segment::unused_space()

os_unsigned_int32 unused_space() const;

Returns the amount of space (in bytes) in the segment not currently occupied by any object. It accounts for space resulting from objects that have been deleted as well as space that cannot be used as a result of internal ObjectStore alignment considerations.

You can use it as a rough guide to determine whether a segment needs to be compacted.

os_segment_access

Instances of the class **os_segment_access** serve to associate zero or more access types with a group of a specified name, as well as with the default group.

By associating an **os_segment_access** with a segment (using **os_segment::set_access_control()**), you specify the segment's associated primary group and the segment's permissions.

The owner of a segment always has both read and write access to it.

The possible combinations of access types are represented by the following enumerators:

- **os_segment_access::no_access**
- **os_segment_access::read_access**
- **os_segment_access::read_write_access**

Note that write access without read access cannot be specified.

These enumerators are used as arguments to some of the members of **os_segment_access**.

You must be the owner of a database to set the permissions on its segments. If you are not the owner of a database but nevertheless have write access to it, you have the ability to create a segment in the database but not to modify its permissions. Since newly created segments allow all types of access to all categories of users, segments created by nonowners necessarily have a period of vulnerability, between creation time and the time at which the owner restricts access with **os_segment::set_access_control()**.

See Chapter 7, Database Access Control, in *ObjectStore C++ API User Guide*.

os_segment_access::get_default()

os_int32 get_default() const;

Returns the types of access associated with the default group for the **os_segment_access**.

os_segment_access::get_primary_group()

```
os_int32 get_primary_group(
    os_char_const_p* group_name = 0
) const;
```

Returns the types of access associated with the primary group of the **os_segment_access**. The function sets **group_name**, if supplied, to point to the name of that group.

os_segment_access::no_access

This is one of three enumerators used to specify combinations of access types. They are used as arguments to some of the members of **os_segment_access**.

os_segment_access::operator =()

```
os_segment_access& operator =(
    const os_segment_access& source
);
```

Modifies the **os_segment_access** pointed to by **this** so that it is a copy of **source**, that is, so that it stores the same group name as **source**, and associates the same combinations of access types with the same groups. It returns a reference to the modified **os_segment_access**.

os_segment_access::os_segment_access()

```
os_segment_access();
```

Creates an **os_segment_access** that associates **no_access** with both the default group and the group named **group_name**.

```
os_segment_access(
    const char* primary_group,
     os_int32 primary_group_access_type,
     os_int32 default_access_type
);
```

Creates an instance of **os_segment_access** that associates **primary_group_access_type** with the group named **primary_group**, and associates **default_access_type** with the default group. **primary_group_access_type** and **default_access_type** are each **os_segment_access::no_access**, **os_segment_access::read_access**, or **os_segment_access::read_write_access**.

```
os_segment_access(
    const os_segment_access& source
);
```

Creates a copy of **source**, that is, it creates an **os_segment_access** that stores the same group name, and associates the same combinations of access types with the same groups.

os_segment_access::read_access

This is one of three enumerators used to specify combinations of access types. They are used as arguments to some of the members of **os_segment_access**.

os_segment_access::read_write_access

This is one of three enumerators used to specify combinations of access types. They are used as arguments to some of the members of **os_segment_access**.

os_segment_access::set_default()

```
void set_default(
    os_int32 access_type
);
```

Associates a specified combination of access types with the default group. **access_type** is **os_segment_access::no_access**, **os_segment_access::read_access**, or **os_segment_access::read_ write_access**.

os_segment_access::set_primary_group()

```
void set_primary_group(
    const char* group_name,
    os_int32 access_type
);
```

Associates a specified combination of access types with a group of a specified name. **group_name** is the name of the group. **access_ type** is **os_segment_access::no_access**, **os_segment_ access::read_access**, or **os_segment_access::read_write_access**.

```
void set_primary_group(
    os_int32 access_type
);
```

Associates a specified combination of access types with a group of an unspecified name. **access_type** is **os_segment_access::no_**

access, **os_segment_access::read_access**, or **os_segment_ access::read_write_access**.

os_segment_access::~os_segment_access()

The destructor frees memory associated with the deleted instance of **os_segment_access**.

os_server

Instances of the class **os_server** represent ObjectStore Servers. This class is useful for handling err_broken_server_connection.

You get pointers to all the Servers currently known to the client by calling **objectstore::get_all_servers()**. Here is an example of a handler for err_broken_server_connection:

```
TIX_HANDLE (err_broken_server_connection) {
/* code that could encounter broken connection */
}
TIX_EXCEPTION {
    tix_exception *cur = tix_handler::get_exception();

    /* It might be necessary to abort a transaction in
        progress. This is a tricky situation since
        under some circumstances it is not needed, and
        in others it is.  */
        if (objectstore::abort_in_progress())
        os_transaction::abort_top_level();

    /* here is how the application can find the lost servers. */
    os_int32 nservers = objectstore::get_n_servers();
    os_server **svrlist = new os_server * [nservers];
    char *svr_name = NULL;
    os_int32 ignore;
    objectstore::get_all_servers(nservers, svrlist, ignore);

    for (os_int32 i = 0; i < nservers; ++i) {
        if (svrlist[i]->connection_is_broken()) {
            svr_name = svrlist[i]->get_host_name();
            printf("lost server %s\n", svr_name);
            delete [] svr_name;
        }
    }
    delete [] svrlist;
} TIX_END_HANDLE;
```

When handling err_broken_server_connection, call **os_transaction::abort_top_level()** if **objectstore::abort_in_progress()** returns nonzero. If you do not abort the transaction, attempts to proceed might cause err_broken_server_connection to be reraised.

When ObjectStore raises err_broken_server_connection, it immediately aborts all transactions. Any databases currently open on the lost Server are considered by ObjectStore to remain open. Subsequent uses of these databases (or others managed by that Server) will cause ObjectStore to attempt to reconnect with the Server.

os_server::connection_is_broken()

os_boolean connection_is_broken();

Returns nonzero if the connection with the specified Server is currently lost; otherwise returns **0**.

os_server::disconnect()

void disconnect();

Disconnects the specified Server. Call this function outside any transaction; otherwise err_trans is signaled. It is harmless to call this member if the connection has already been lost for any reason. The result of this call is very much like the result of unintentionally losing a connection. The client retains all its information about the Server and its databases, but marks them as having lost the connection. An attempt to access a database on the Server will cause ObjectStore to attempt to reconnect to the Server. You can also call **os_server::reconnect()** on the Server.

os_server::get_databases()

```
void get_databases(
    os_int32 max_to_return,
    os_database_p * dbs,
    os_int32& n_ret
);
```

Provides access to all the databases associated with the specified Server, whether open or closed. The **os_database_p*** is an array of pointers to **os_database** objects. This array must be allocated by the user. The function **os_server::get_n_databases()** can be used to determine how large an array to allocate. **max_to_return** is specified by the user, and is the maximum number of elements the array is to have. **n_ret** refers to the actual number of elements in the array.

os_server::get_host_name()

char *get_host_name() const;

Returns the name of the host of the specified Server. It is the caller’s responsibility to deallocate the string when it is no longer needed.

char* get_host_name();

For failover Servers, this function returns the logical failover Server host name. Note that the logical Server name is not always identical to the Server name for the machine providing access to the database. The caller should delete the returned value. See **os_failover_server::get_online_server()**.

os_server::get_n_databases()

os_int32 get_n_databases();

Returns the number of databases associated with the specified Server, whether open or closed.

os_server::is_failover()

os_boolean is_failover() const;

Returns true if and only if the **os_server*** is an **os_failover_server** also.

This method is used to identify the **os_failover_server** in the list of Servers returned by **objectstore::get_all_servers()**.

os_server::reconnect()

void reconnect();

Causes ObjectStore to immediately attempt to reconnect to the specified Server. Note that exceptions, such as err_server_refused_connection, might result. Calling this function has no effect if the connection is not currently broken.

os_str_conv

This class provides conversion facilities for various Japanese text encoding methods: EUC, JIS, SJIS, Unicode, and UTF8.

It provides a facility, called autodetect, to detect the encoding of a given string. This is useful for applications in which a client might send strings in an unknown format, a common issue for Internet applications.

os_str_conv::change_mapping()

int change_mapping(mapping table[],size_t table_sz);

You can modify the mapping behavior of an existing instance of **os_str_conv** (whether heap- or stack-allocated) by calling **os_str_conv::change_mapping()**. Override information is stored for future conversion services associated with that instance.

The override mapping information applies to whatever explicit mapping has been established for the given **os_str_conv** instance. Mappings of **os_str_conv** instances cannot be overridden by instances using autodetect. Attempts to do so will return **-1** from **change_mapping()** to indicate this error condition.

The **change_mapping()** method takes the following two parameters:

mapping_table[]
An array of mapping code pairs that can be allocated locally, globally, or on the heap. If the array is heap-allocated, the user must delete it after calling **change_mapping()**.

Internally, **change_mapping()** makes a sorted copy of **mapping_table[]**. The sorting provides quick lookup at run time. The internal copy is freed when the **os_str_conv** destructor is eventually called.

Note that the mapping pairs are unsigned 32-bit quantities. The LSB (least significant bit) is on the right, so, for example, the single-byte character 0x5C is represented as 0x0000005C, and the two-byte code 0x81,0x54 is 0x0000815F.

size_t table_sz
The number of elements in the **mapping_table**. Be sure that this is not the number of bytes in the array.

os_str_conv::convert()

Since Unicode is a 16-bit quantity, byte order depends on platform architecture. (Other encodings are byte streams and therefore do not depend on processor architecture.) On little-endian systems, such as Intel, the low-order byte comes first. On big-endian systems (Sparc, HP, and Mips, for example) the high-order byte is first.

There are three overloadings to the **os_str_conv::convert()** method to provide flexibility for dealing with Unicode strings with different byte-ordering schemes. If a parameter is of **char*** type, all 16-bit quantities are considered big-endian, regardless of platform. However, if the type is **os_unsigned_int16*,** the values assigned or read are handled according to the platform architecture.

encode_type convert(char* dest, const char* src);

If either **dest** or **src** is a buffer containing Unicode characters, these 16-bit characters are considered big-endian, regardless of platform architecture.

encode_type convert(os_unsigned_int16* dest, const char* src);

This overloading interprets 16-bit Unicode buffer **dest** according to the byte order of the processor used.

encode_type convert(char* dest, const os_unsigned_int16* src);

This overloading interprets 16-bit Unicode buffer **src** according to the byte order of the processor used.

os_str_conv::get_converted_size()

virtual size_t get_converted_size(const char* src) const;

Returns the size of the buffer, in units of bytes, required to contain the converted result of the given **src** string. If **src** is a Unicode string, its 16-bit characters are considered big-endian, regardless of platform architecture.

Because the entire source string must be examined, the time it takes for this function to complete is proportional to the length of the source string.

If the autodetect mode is used and autodetect fails to determine the encoding of **src**, **get_converted_size()** returns **0**.

```
virtual size_t get_converted_size(
    const os_unsigned_int16* src
) const;
```

Returns the size of the buffer, in units of bytes, required to contain the converted result of the given **src** string. If **src** is a Unicode string, its 16-bit characters are interpreted according to the byte order of the processor used.

Because the entire source string must be examined, the time it takes for this function to complete is proportional to the length of the source string.

If the autodetect mode is used and autodetect fails to determine the encoding of **src**, **get_converted_size()** returns **0**.

os_str_conv::os_string_conv()

```
os_str_conv(
    encode_type_enum dest,
    encode_type_enum src=AUTOMATIC
);
```

Instantiates a conversion path.

encode_type_enum can be one of

AUTOMATIC	Determine the encoding of the source string automatically. This automatic detection distinguishes EUC and SJIS only. It might not correctly detect SJIS if the string contains half-width kana.
AUTOMATIC_ALLOW_KANA	Determine the encoding of the source string automatically. This automatic detection distinguishes EUC and SJIS only. This correctly interprets SJIS strings that contain half-width kana, but it might incorrectly interpret certain EUC strings as SJIS strings.
ASCII	Strings are interpreted as single-byte ASCII characters.
SJIS	Strings are interpreted as multibyte Japanese strings of SJIS encoding.

EUC	Strings are interpreted as multibyte Japanese strings of EUC encoding.
UNICODE	Strings are interpreted as Japanese strings of Unicode encoding.
JIS	Strings are interpreted as multibyte Japanese strings of JIS encoding.
UTF8	Strings are interpreted as multibyte Japanese strings of UTF-8 encoding.

os_subscription

Objects of class **os_subscription** are created by users in order to perform subscription and unsubscription operations. Note that you do not always have to create **os_subscription** objects in order to subscribe or unsubscribe. You can accomplish a single subscription or unsubscription by passing an **os_database**, **os_segment**, **os_object_cluster,** or address range directly to **os_notification::subscribe()** or **os_notification::unsubscribe()**.

The main reason to manipulate **os_subscription** objects directly is to pass an array of them to **os_notification::subscribe**. There are overloadings of **os_notification::subscribe** that take the same sets of arguments as **os_subscription** constructors. Use these if you want to subscribe only to a single notification address. You can call these directly and completely bypass the use of **os_subscription**s.

The reason you might want to pass an array of **os_subscription** to **os_notification::subscribe** is that it is much more efficient to call **os_notification::subscribe** once with an array than to call it separately for each subscription when there are multiple subscriptions to register.

Each **os_subscription** object represents an address range in an ObjectStore database. There are four constructors that allow creation of subscriptions covering an entire database, an entire segment, an entire object cluster, or a specific location range:

os_subscription::os_subscription()

There is a default constructor, **os_subscription();**, that creates an *uninitialized* subscription.

os_subscription(const os_database *);

This constructor is used to create a subscription to an entire database.

os_subscription(const os_segment *);

This constructor is used to create a subscription to a segment.

os_subscription(const os_object_cluster *);

This constructor is used to create a subscription to an object cluster.

```
os_subscription(
const os_reference &,
os_int32 n_bytes = 1
);
```

os_subscription::assign()

The default constructor is most useful when you are allocating an array of subscriptions. Each **os_subscription** in the array will initially be uninitialized. Each array element can then be initialized using the **assign** or **operator=** member functions:

```
void assign(const os_database *);
void assign(const os_segment *);
void assign(const os_object_cluster *);
void assign(const os_reference &, os_int32 n_bytes = 1);

os_subscription &operator=(const os_database *db);
os_subscription &operator=(const os_segment *seg);
os_subscription &operator=(const os_object_cluster *clus);
os_subscription &operator=(const os_reference &ref);
```

Objects of type **os_subscription** can be reassigned in this fashion as many times as desired. Note that because **operator=** only allows a single argument, you must use the final form of **assign** if you want to specify **n_bytes > 1**.

When passing database locations to **os_subscription** member functions, you do not need to explicitly convert to **os_reference**. You can pass pointers or **os_Reference<X>**; these are converted by C++ automatically.

os_subscription::get_database()

```
os_database *get_database();
```

The only accessor for **os_subscription** returns the database associated with the subscription. An uninitialized **os_subscription** has a null (0) database.

Subscribe and unsubscribe nonstatic member functions are provided as shortcuts for calling **os_notification::subscribe()** and **os_notification::unsubscribe()**. They are

- **void subscribe();**
- **void unsubscribe();**

Further discussion

See **os_namespace** on page 175 for further discussion about notification.

os_template

This class is part of the ObjectStore metaobject protocol, which provides access to ObjectStore schemas. Instances of this class represent (type or function) templates. **os_type_template** and **os_function_template** are derived from **os_template**.

os_template::Function

This enumerator is a possible return value from **os_template::get_kind()**, indicating a function template.

os_template::get_args()

os_List<const os_template_formal_arg*> get_args() const;

Returns a list (in declaration order) of the formal arguments of the specified template. Each element of the list is a pointer to a **const os_template_formal_arg**.

os_template::get_kind()

enum os_template_kind { Type, Function } ;

os_template_kind get_kind() const;

Returns an enumerator indicating whether the specified template is a type template or a function template. The possible return values are **os_template::Type** and **os_template::Function**.

os_template::is_unspecified()

os_boolean is_unspecified() const;

Returns nonzero (that is, true) if and only if the specified **os_template** is the *unspecified template*. Some **os_template**-valued attributes in the metaobject protocol are required to have values in a consistent schema, but might lack values in the transient schema, before schema installation or evolution is performed. The get function for such an attribute returns a reference to an **os_template**. The fact that a reference rather than pointer is returned indicates that the value is required in a consistent schema. In the transient schema, if such an attribute lacks a value (because you have not yet specified it), the get function returns the unspecified template. This is the only **os_template** for which **is_unspecified()** returns nonzero.

os_template::operator const os_type_template&()

operator const os_type_template&() const;

Provides for safe conversion to **const os_type_template**. If the specified **os_template** is not an **os_type_template**, err_mop_illegal_cast is signaled.

os_template::operator os_type_template&()

operator os_type_template&();

Provides for safe conversion to **os_type_template**. If the specified **os_template** is not an **os_type_template**, err_mop_illegal_cast is signaled.

os_template::set_args()

void set_args(os_List<os_template_formal_arg*>&);

Specifies the list (in declaration order) of the formal arguments of the specified template. Each element of the list is a pointer to an **os_template_formal_arg**.

os_template::Type

This enumerator is a possible return value from **os_template::get_kind()**, indicating a type template.

os_template_actual_arg

This class is part of the ObjectStore metaobject protocol, which provides access to ObjectStore schemas. Instances of this class represent actual arguments used to instantiate (type or function) templates. The types **os_type_template_actual_arg** and **os_literal_template_actual_arg** are derived from **os_template_actual_arg**.

os_template_actual_arg::get_kind()

enum os_template_actual_arg_kind { type_actual, literal_actual } ;

os_template_actual_arg_kind get_kind() const;

Returns an enumerator indicating whether the specified actual is a type or value (that is, literal).

os_template_actual_arg::operator const os_literal_template_actual_arg&()

operator const os_literal_template_actual_arg&() const;

Provides for safe conversion to **const os_literal_template_actual_arg&**. If the specified **os_template_actual_arg** is not an **os_literal_template_actual_arg**, err_mop_illegal_cast is signaled.

os_template_actual_arg::operator const os_type_template_actual_arg&()

operator const os_type_template_actual_arg&() const;

Provides for safe conversion to **const os_type_template_actual_arg&**. If the specified **os_template_actual_arg** is not an **os_type_template_actual_arg**, err_mop_illegal_cast is signaled.

os_template_actual_arg::operator os_literal_template_actual_arg&()

operator os_literal_template_actual_arg&();

Provides for safe conversion to **os_literal_template_actual_arg&**. If the specified **os_template_actual_arg** is not an **os_literal_template_actual_arg**, err_mop_illegal_cast is signaled.

os_template_actual_arg::operator os_type_template_actual_arg&()

operator os_type_template_actual_arg&();

Provides for safe conversion to **const os_type_template_actual_arg&**. If the specified **os_template_actual_arg** is not an **os_type_template_actual_arg**, err_mop_illegal_cast is signaled.

os_template_formal_arg

This class is part of the ObjectStore metaobject protocol, which provides access to ObjectStore schemas. Instances of this class represent formal arguments for (type or function) templates. This class has no public members.

os_template_formal_arg::get_kind()

enum os_template_formal_arg_kind { Type, Value };

os_template_formal_arg_kind get_kind() const;

Returns an enumerator indicating whether the specified formal is a type parameter or value parameter (that is, whether its actuals are types or values).

os_template_formal_arg::get_name()

const char* get_name() const;

Returns the name of the specified formal parameter.

os_template_formal_arg::operator const os_template_type_formal&()

operator const os_template_type_formal&() const;

Provides for safe conversion to **const os_template_type_formal&**. If the specified **os_template_formal_arg** is not an **os_template_type_formal**, err_mop_illegal_cast is signaled.

os_template_formal_arg::operator const os_template_value_formal&()

operator const os_template_value_formal&()const;

Provides for safe conversion to **const os_template_value_formal&**. If the specified **os_template_formal_arg** is not an **os_template_value_formal**, err_mop_illegal_cast is signaled.

os_template_formal_arg::operator os_template_type_formal&()

operator os_template_type_formal&();

Provides for safe conversion to **os_template_type_formal&**. If the specified **os_template_formal_arg** is not an **os_template_type_formal**, err_mop_illegal_cast is signaled.

os_template_formal_arg::operator os_template_value_formal&()

operator os_template_value_formal&();

Provides for safe conversion to **os_template_value_formal&**. If the specified **os_template_formal_arg** is not an **os_template_value_ formal**, err_mop_illegal_cast is signaled.

os_template_formal_arg::set_name()

void set_name (const char *name);

Sets the name of the specified formal parameter.

os_template_instantiation

This class is part of the ObjectStore metaobject protocol, which provides access to ObjectStore schemas. Instances of this class represent instantiations of a (type or function) template. The class **os_type_template** is derived from **os_template_instantiation**.

os_template_instantiation::create()

```
static os_template_instantiation& create(
    os_template*,
    os_List<os_template_actual_arg*>*
);
```

Creates an **os_template_instantiation** from the specified template and actual parameters.

os_template_instantiation::get_args()

```
os_List<const os_template_actual_arg*> get_args() const;
```

Returns a list (in declaration order) of the actual arguments used to instantiate the associated template. Each element of the list is a pointer to a **const os_template_actual_arg**.

```
os_List<os_template_actual_arg*> get_args();
```

Returns a list (in declaration order) of the actual arguments used to instantiate the associated template. Each element of the list is a pointer to an **os_template_actual_arg**.

os_template_instantiation::get_template()

```
const os_template &get_template() const;
```

Returns the **const** template instantiated by the specified **os_ template_instantiation**.

```
os_template &get_template();
```

Returns the non-**const** template instantiated by the specified **os_ template_instantiation**.

os_template_instantiation::set_args()

```
void set_args(os_List<os_template_actual_arg*>&);
```

Sets (in declaration order) the actual arguments with which to instantiate the associated template.

os_template_instantiation::set_template()

void set_template(os_template&);

Sets the template instantiated by the specified **os_template_ instantiation**.

os_template_type_formal

class os_template_type_formal : public os_template_formal_arg

This class is part of the ObjectStore metaobject protocol, which provides access to ObjectStore schemas. Instances of this class represent formal template parameters whose actuals are types. The class **os_template_type_formal** is derived from **os_template_formal_arg**.

os_template_type_formal::create()

static os_template_type_formal& create(const char *name);

Creates an **os_template_type_formal** with the specified name.

os_template_value_formal

class os_template_value_formal : public os_template_formal_arg

This class is part of the ObjectStore metaobject protocol, which provides access to ObjectStore schemas. Instances of this class represent formal template parameters whose actuals are values. The class **os_template_value_formal** is derived from **os_template_formal_arg**.

os_template_value_formal::create()

```
static os_template_value_formal& create(
    const char *name,
    os_type *type
);
```

Creates an **os_template_value_formal** with the specified name. The actuals for the created formal are instances of the specified type.

os_template_value_formal::get_type()

const os_type& get_type() const;

Returns a **const** reference to an **os_type** representing the type of the actuals for the specified formal.

os_type& get_type();

Returns a non-**const** reference to an **os_type** representing the type of the actuals for the specified formal.

os_template_value_formal::set_type()

void set_type(os_type&);

Sets the type of the actuals for the specified formal.

os_transaction

Instances of the class **os_transaction** represent transactions of the current process.

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.

All ObjectStore programs must include the header file **<ostore/ostore.hh>**.

os_transaction::abort()

static void abort(os_transaction* = get_current());

Aborts the specified transaction. For dynamic transactions, control flows to the next statement after you call **abort()**. For lexical transactions, control flows to the next statement after the end of the current transaction block. Persistent data is rolled back to its state as of the beginning of the transaction. In addition, if the aborted transaction is not nested within another transaction, all locks are released, and other processes can access the pages that the aborted transaction accessed.

os_transaction::abort_only

This enumerator is an optional parameter used in the transaction macro to specify a transaction that can write to persistent data, but cannot be committed. This enumeration can be used in place of **os_transaction::update** or **os_transaction::read_only**. This provides the client with the capability to write to databases that are

- Read-only
- Protected against writes
- MVCC-opened

For clients to take advantage of **abort_only** transactions, the Cache Manager has to be at ObjectStore 4.0.2 release level or greater, and every Server connected to must also be at the 4.0.2 level. This restriction is only enforced when the application attempts to use **abort_only** transactions.

There are no write locks for any pages that are written during an abort-only transaction. This allows multiple concurrent abort-only writers to a database. However, there are read locks for all pages the client reads or writes. When used with MVCC-opened databases, the standard MVCC locks apply.

The client raises the exception err_commit_abort_only if an attempt is made to commit an abort-only transaction. If the top-level transaction is abort-only, both abort-only and update transactions can nest within it. Otherwise, the nesting rule described in **os_transaction::begin()** applies.

Note that an abort_only transaction does not automatically abort. You must specifically use the **os_transaction::abort()** function to abort the abort_only transaction, otherwise an exception is signaled.

You can use **abort_top_level(),** or for stack transactions use **abort()**, since you know exactly where the transaction ends. For example:

```
OS_BEGIN_TXN(txn, 0, os_transaction::abort_only) {
    ...
    os_transaction::abort();
} OS_END_TXN(txn);
```

os_transaction::abort_top_level()

static void abort_top_level();

Aborts the outermost transaction within which control currently resides. If the current transaction is not nested, this function aborts the current transaction.

os_transaction::begin()

```
static os_transaction *begin(
    transaction_type_enum t_type = update,
    transaction_scope_enum t_scope = local);
```

Begins a dynamic transaction. A pointer to an object representing the transaction is returned. The user is responsible for deleting this object after terminating the transaction. The **os_int32** argument should be coded as either **os_transaction::update** or **os_transaction::read_only**, depending on the type of transaction desired. Unlike transactions started with a transaction macro or statement, dynamic transactions are not automatically retried when aborted because of deadlock. Unless the top-level transaction is **abort-only**, a nested transaction must be of the same type (**os_transaction::update** or **os_transaction::read_only**) as its parent; otherwise err_trans_wrong_type is signaled.

To support multithreaded applications, dynamic transactions can be either *local* or *global*. By default, dynamic transactions are local. You start a global transaction by passing the enumerator **os_ transaction::global** as the second argument to **os_ transaction::begin()**.

The two kinds of transactions have the following characteristics:

- Local transaction: a thread enters a local transaction by calling **os_transaction::begin()** or **OS_BEGIN_TXN()**. When one thread enters a local transaction, this has no effect on whether other threads are within a transaction.
- Global transaction: a thread enters a global transaction when it calls **os_transaction::begin()** or when another thread of the same process calls **os_transaction::begin()**. When one thread enters a global transaction by calling **os_transaction::begin()**, all other threads automatically enter the same transaction.

Local transactions synchronize access to the ObjectStore run time by serializing the transactions of the different threads (that is, by making the transactions run one after another without overlapping). After one thread starts a local transaction, if another thread attempts to start a transaction or enter the ObjectStore run time, it is blocked until the local transaction completes. So two threads cannot be in a local transaction at the same time.

Global transactions allow for a somewhat higher degree of concurrency. After one thread enters the ObjectStore run time, if another thread attempts to enter the ObjectStore run time, it is blocked until control in the first thread exits from the run time. Although two threads cannot be in the ObjectStore run time at the same time, there can be some interleaving of operations of different threads within a transaction.

If you use global transactions, be sure to synchronize the threads so that no thread attempts to access persistent data while another thread is committing or aborting. Place a barrier before the end of the transaction so that all participating threads complete work on persistent data before the end-of-transaction operation is allowed to proceed.

You cannot nest a local transaction within a global transaction, nor can you nest a global transaction within a local one. This table specifies how two transactions can interact:

	Thread A tries global txn	***Thread A tries local txn***	***Thread B tries global txn***	***Thread B tries local txn***
Thread A runs global txn	OK, nested global txn	err_trans_ wrong_type is signaled	OK, nested global txn	err_trans_ wrong_type is signaled
Thread A runs local txn	err_trans_ wrong_type is signaled	OK, nested local txn	OK, but block until A completes	OK, but block until A completes

```
static os_transaction *os_transaction::begin(
    transaction_type_enum t_type = update,
    transaction_scope_enum t_scope = local,
    os_transaction *parent
);
```

Like the previous overloading, except that the new transaction is nested within the specified transaction.

```
static os_transaction *os_transaction::begin(
    char *name,
    transaction_type_enum t_type = update,
    transaction_scope_enum t_scope = local);
```

Like the first overloading, except that the new transaction has the specified name.

```
static os_transaction *os_transaction::begin(
    char *name,
    transaction_type_enum t_type = update,
    transaction_scope_enum t_scope = local,
    os_transaction *parent
);
```

Like the first overloading, except that the new transaction has the specified name and is nested within the specified transaction.

os_transaction::checkpoint()

Note: Like transaction commit and abort, checkpoint operations are not thread-safe. Applications must ensure that other threads do not access persistent memory during a checkpoint operation.

```
static void checkpoint();
```

Invokes checkpoint on the current transaction.

static checkpoint(os_transaction*);

Invokes checkpoint on the given transaction. Note that checkpoint is only valid for a top-level dynamic transaction. Attempts to checkpoint nested or stack transactions result in an exception's being thrown.

os_transaction::checkpoint_in_progress();

os_boolean os_transaction::checkpoint_in_progress();

Returns **1** if a checkpoint is currently in progress. This function is for use in hook functions registered for invocation during transaction commit processing.

os_transaction::commit()

static void commit(os_transaction* = get_current());

Commits the specified dynamic transaction. Unlike transactions started with a transaction statement, dynamic transactions are not automatically retried when aborted because of deadlock.

os_transaction::exception_status

tix_exception *exception_status;

ObjectStore stores an **exception*** in this member if the specified transaction is aborted because of the raising of an exception. The stored **exception*** indicates the exception that caused the abort. ObjectStore stores **0** in this location at the beginning of each transaction.

Raising an exception will cause a transaction to abort if the exception is handled outside the transaction's dynamic scope, or if there is no handler for the exception.

os_transaction::get_current()

static os_transaction *get_current();

Returns a pointer to the most deeply nested transaction in which control currently resides. The value is **0** if no transaction is in progress.

os_transaction::get_max_retries()

static os_int32 get_max_retries();

Returns, for the current process, the number of times a transaction is automatically retried after being aborted because of deadlock. (Note that this does not apply to dynamic transactions, which are not automatically retried.)

os_transaction::get_name()

char *get_name() const;

Returns the name of the specified transaction, as set by **os_ transaction::set_name()**. It is the caller's responsibility to deallocate the string when it is no longer needed.

os_transaction::get_parent()

os_transaction *get_parent() const;

Returns a pointer to the transaction within which the specified transaction is directly nested.

os_transaction::get_scope()

os_transaction_scope_enum get_scope() const;

Returns **os_transaction::local** or **os_transaction::global**, depending on which is true of the current transaction.

os_transaction::get_type()

os_transaction_type get_type() const;

Returns **os_transaction::abort-only**, **os_transaction::read_only,** or **os_transaction::update**, depending on which is true of the current transaction.

os_transaction::is_aborted()

os_boolean is_aborted() const;

Returns nonzero if the specified dynamic transaction is aborted, and **0** otherwise. For a transaction newly created by **os_ transaction::begin()**, **0** is returned.

os_transaction::is_committed()

os_boolean is_committed() const;

Returns nonzero if the specified dynamic transaction is committed, and **0** otherwise. For a transaction newly created by **os_transaction::begin()**, **0** is returned.

os_transaction::is_prepare_to_commit_completed()

is_prepare_to_commit_completed();

Returns true if a **prepare_to_commit** was invoked and completed in the current transaction and false otherwise.

os_transaction::is_prepare_to_commit_invoked()

is_prepare_to_commit_invoked();

Returns true if a **prepare_to_commit** was invoked in the current transaction and false otherwise.

os_transaction::prepare_to_commit()

prepare_to_commit();

Performs, in advance, the parts of transaction commit that can fail due to the inability to acquire a resource. If this method successfully completes, the actual transaction commit is virtually reduced to sending a commit message to the ObjectStore Server. All the modified data was sent to the Server during the invocation of the **prepare_to_commit** call. After the call to **prepare_to_commit**, the only ObjectStore operations that should occur are transaction commit or abort.

Some of the failures that can occur during the call to **prepare_to_commit()** are

- Client's being selected as a deadlock victim by the Server. The exception err_deadlock would be raised.
- Server's running out of disk space while growing a segment or writing to the log file. The exception err_server_full would be raised.
- The exception err_broken_server_connection can occur if the Server fails during commit.

Restrictions on use:

The exception err_prepare_to_commit is generated if

- Access to persistent data after a call to **prepare_to_commit**.

- Any **objectstore** operation other than commit or abort is attempted after a call to **prepare_to_commit**.
- The method is invoked within a nested transaction.

Once err_prepare_to_commit is handled, the transaction cannot do anything else with ObjectStore and it must call **os_ transaction::abort_top_level()**.

os_transaction::read_only

This enumerator is an optional parameter used in the transaction macro to specify a transaction that performs no updates on persistent storage.

os_transaction::set_max_retries()

static void set_max_retries(os_int32);

Sets, for the current process, the number of times a transaction is automatically retried after being aborted because of deadlock.

os_transaction::set_name()

void set_name(const char *new_name);

Sets the name of the specified transaction. The function copies the string **new_name**.

os_transaction::top_level()

os_boolean top_level() const;

Returns nonzero if the specified transaction is nonnested; returns **0** otherwise.

os_transaction::update

This enumerator is an optional parameter used in the transaction macro to specify a transaction that performs updates on persistent storage.

os_transaction_hook

This class provides functions for registering and deregistering transaction hook functions. It also provides enumerators for specifying the type of event to trigger invocation of a given hook function.

Programs using this class must include the header file **<ostore/tranhook.hh>** after including **<ostore/ostore.hh>**.

os_transaction_hook::after_begin

This enumerator is used as an argument to **register_hook()** to specify that a hook function is to be invoked just after each transaction begins.

os_transaction_hook::after_commit

This enumerator is used as an argument to **register_hook()** to specify that a hook function is to be invoked just after each transaction commit.

os_transaction_hook::before_abort

This enumerator is used as an argument to **register_hook()** to specify that a hook function is to be invoked just before each transaction abort.

os_transaction_hook::before_commit

This enumerator is used as an argument to **register_hook()** to specify that a hook function is to be invoked just before each transaction commit.

os_transaction_hook::before_retry

This enumerator is used as an argument to **register_hook()** to specify that a hook function is to be invoked just before each retry of an aborted transaction.

os_transaction_hook::register_hook()

```
typedef void (*hook_t)(
    const os_int32 event_type, const os_transaction*
    );
```

Solaris SunPro

```
typedef void (*hook_t)(
    const os_transaction_hook::event_type, const os_transaction*
    );
```

```
static hook_t register_hook(
    os_int32 event_type, hook_t hook_fn
    );
```

Solaris SunPro

```
static hook_t register_hook(
    os_transaction_hook::event_type, hook_t hook_fn
    );
```

Registers **hook_fn** and specifies that it should be called each time an event of type **event_type** occurs. **event_type** is one of the following enumerators:

- **os_transaction_hook::after_begin**
- **os_transaction_hook::after_commit**
- **os_transaction_hook::before_abort**
- **os_transaction_hook::before_commit**
- **os_transaction_hook::before_retry**

A pointer to the current hook function is returned, or **0** if there is none. The application must ensure that any previously registered hook functions, as returned by **register_hook()**, are called at some point during the execution of the current hook function.

When **hook_fn** is invoked, the arguments passed to it are as follows. The first argument (**event_type**) is an enumerator indicating the type of event that triggered invocation. The second argument is a pointer to the current transaction. If the first argument is **os_transaction_hook::before_retry**, the second argument is **0**.

Do not abort or commit the current transaction from within a hook function.

os_transaction_hook::deregister_hook()

```
static void deregister_hook(os_int32 event_type);
```

Deregisters all hook functions with event type **event_type**. Note that if a previously registered hook function was returned by **register_hook()**, it must be reregistered.

os_transformer_binding

Instances of this class represent an association between a class and a transformer function. Instances of **os_transformer_binding** are used as arguments to **os_schema_evolution::augment_post_evol_ transformers()**. Instances should be allocated in transient memory only.

os_transformer_binding::os_transformer_binding()

```
os_transformer_binding(
    char *class_name,
    void (*f)(void*)
);
```

Associates the class named **class_name** with the function **f**.

os_type

This class is part of the ObjectStore metaobject protocol, which provides access to ObjectStore schemas. Instances of this class represent C++ types.

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.

Programs using this class must include **<ostore/ostore.hh>**, followed by **<ostore/coll.hh>** (if used), followed by **<ostore/mop.hh>**.

os_type::Anonymous_indirect

This enumerator indicates a **const** or **volatile** type. It is a possible return value from **os_type::get_kind()**.

os_type::Array

This enumerator indicates an array type. It is a possible return value from **os_type::get_kind()**.

os_type::Char

This enumerator indicates a character type. It is a possible return value from **os_type::get_kind()**.

os_type::Class

This enumerator indicates a class. It is a possible return value from **os_type::get_kind()**.

os_type::Double

This enumerator indicates the type **double**. It is a possible return value from **os_type::get_kind()**.

os_type::Enum

This enumerator indicates an enumeration type. It is a possible return value from **os_type::get_kind()**.

os_type::Float

This enumerator indicates the type **float**. It is a possible return value from **os_type::get_kind()**.

os_type::Function

This enumerator indicates a function type. It is a possible return value from **os_type::get_kind()**.

os_type::get_alignment()

os_unsigned_int32 get_alignment() const;

Gets the alignment associated with the type.

os_type::get_enclosing_class()

const os_class_type *get_enclosing_class() const;

If a class's definition is nested within that of another class, this other class is the *enclosing class* of the nested class. The function returns a **const** pointer to the enclosing class, or **0** if there is no enclosing class.

os_class_type *get_enclosing_class();

If a class's definition is nested within that of another class, this other class is the *enclosing class* of the nested class. The function returns a non-**const** pointer to the enclosing class, or **0** if there is no enclosing class.

os_type::get_kind()

os_unsigned_int32 get_kind() const;

Returns one of the following enumerators describing the specified instance of **os_type**: **Type**, **Void**, **Unsigned_char**, **Signed_char**, **Char**, **Wchar_t**, **Unsigned_short**, **Signed_short**, **Integer**, **Unsigned_integer**, **Signed_long**, **Unsigned_long**, **Float**, **Double**, **Long_double**, **Named_ indirect**, **Anonymous_indirect**, **Pointer**, **Reference**, **Pointer_to_ member**, **Array**, **Class**, **Instantiated_class**, **Enum**, **Function**, or **Undefined**. These enumerators are defined within the scope of the class **os_type**.

os_type::get_kind_string()

static const char *get_kind_string(os_type_kind);

Returns a string representation of a type kind, as specified with one of the enumerators returned by **get_kind()**.

os_type::get_size()

os_unsigned_int32 get_size();

Returns the size of the specified type in bytes.

os_type::get_string()

char *get_string() const;

Returns a new string containing a printable representation of the specified instance of **os_type**. The string must be deleted after its value has been consumed.

os_type::Instantiated_class

This enumerator indicates a class that is an instantiation of a class template. It is a possible return value from **os_type::get_kind()**.

os_type::Integer

This enumerator indicates the type **int**. It is a possible return value from **os_type::get_kind()**.

os_type::is_class_type()

os_boolean os_type::is_class_type() cons;t

Returns **1** if the type is an **os_class_type** or **os_instantiated_class_ type.**

os_type::is_pointer_type()

os_boolean os_type::is_pointer_type() const;

Returns **1** if the type is an **os_pointer_type**, **os_reference_type**, or **os_pointer_to_member_type.**

os_type::is_const()

os_boolean is_const() const;

Returns **1** if and only if the type expression associated with the specified instance of **os_type** includes a **const** type specifier.

os_type::is_indirect_type()

os_boolean os_type::is_indirect_type() const;

Returns true if the type is an **os_anonymous_indirect_type** or **os_named_indirect_type.**

os_type::is_integral_type()

os_boolean is_integral_type() const;

Returns nonzero if the specified **os_type** is an instance of **os_integral_type** (such as one representing the type **unsigned int**). Returns **0** otherwise.

os_type::is_real_type()

os_boolean is_real_type() const;

Returns nonzero if the specified **os_type** is an instance of **os_real_type** (such as one representing the type **long double**). Returns **0** otherwise.

os_type::is_unspecified()

os_boolean is_unspecified() const;

Returns nonzero if and only if the specified **os_type** is the *unspecified type*. Some **os_type-**valued attributes in the metaobject protocol are required to have values in a consistent schema, but might lack values in the transient schema, before schema installation or evolution is performed. The get function for such an attribute returns a reference to an **os_type** or **os_class_type**. The fact that a reference rather than a pointer is returned indicates that the value is required in a consistent schema. In the transient schema, if such an attribute lacks a value (because you have not yet specified it), the get function returns the unspecified type. This is the only **os_type** for which **is_unspecified()** returns nonzero.

os_type::is_volatile()

os_boolean is_volatile() const;

Returns **1** if and only if the type expression associated with the specified instance of **os_type** includes a **volatile** type specifier.

os_type::Long_double

This enumerator indicates the type **long double**. It is a possible return value from **os_type::get_kind()**.

os_type::Named_indirect

This enumerator indicates a typedef. It is a possible return value from **os_type::get_kind()**.

os_type::operator const os_anonymous_indirect_type&()

operator const os_anonymous_indirect_type&() const;

Provides for safe casts from **const os_type** to **const os_ anonymous_indirect_type&**. If the cast is not permissible, err_mop_ illegal_cast is signaled.

os_type::operator const os_array_type&()

operator const os_array_type&() const;

Provides for safe casts from **const os_type** to **const os_array_ type&**. If the cast is not permissible, err_mop_illegal_cast is signaled.

os_type::operator const os_class_type&()

operator const os_class_type&() const;

Provides for safe casts from **const os_type** to **const os_class_ type&**. If the cast is not permissible, err_mop_illegal_cast is signaled.

os_type::operator const os_enum_type&()

operator const os_enum_type&() const;

Provides for safe casts from **const os_type** to **const os_enum_ type&**. If the cast is not permissible, err_mop_illegal_cast is signaled.

os_type::operator const os_function_type&()

operator const os_function_type&() const;

Provides for safe casts from **const os_type** to **const os_function_ type&**. If the cast is not permissible, err_mop_illegal_cast is signaled.

os_type::operator const os_instantiated_class_type&()

operator const os_instantiated_class_type&() const;

Provides for safe casts from **const os_type** to **const os_ instantiated_class_type&**. If the cast is not permissible, err_mop_ illegal_cast is signaled.

os_type::operator const os_integral_type&()

operator const os_integral_type&() const;

Provides for safe casts from **const os_type** to **const os_integral_type&**. If the cast is not permissible, err_mop_illegal_cast is signaled.

os_type::operator const os_named_indirect_type&()

operator const os_named_indirect_type&() const;

Provides for safe casts from **const os_type** to **const os_named_indirect_type&**. If the cast is not permissible, err_mop_illegal_cast is signaled.

os_type::operator const os_pointer_type&()

operator const os_pointer_type&() const;

Provides for safe casts from **const os_type** to **const os_pointer_type&**. If the cast is not permissible, err_mop_illegal_cast is signaled.

os_type::operator const os_pointer_to_member_type&()

operator const os_pointer_to_member_type&() const;

Provides for safe casts from **const os_type** to **const os_pointer_to_member_type&**. If the cast is not permissible, err_mop_illegal_cast is signaled.

os_type::operator const os_real_type&()

operator const os_real_type&() const;

Provides for safe casts from **const os_type** to **const os_real_type&**. If the cast is not permissible, err_mop_illegal_cast is signaled.

os_type::operator const os_reference_type&()

operator const os_reference_type&() const;

Provides for safe casts from **const os_type** to **const os_reference_type&**. If the cast is not permissible, err_mop_illegal_cast is signaled.

os_type::operator const os_type_type&()

operator const os_type_type&() const;

Provides for safe casts from **const os_type** to **const os_type_type&**. If the cast is not permissible, err_mop_illegal_cast is signaled.

os_type::operator const os_void_type&()

operator const os_void_type&() const;

Provides for safe casts from **const os_type** to **const os_void_type&**. If the cast is not permissible, err_mop_illegal_cast is signaled.

os_type::operator os_anonymous_indirect_type&()

operator os_anonymous_indirect_type&();

Provides for safe casts from **os_type** to **os_anonymous_indirect_type&**. If the cast is not permissible, err_mop_illegal_cast is signaled.

os_type::operator os_array_type&()

operator os_array_type&();

Provides for safe casts from **os_type** to **os_array_type&**. If the cast is not permissible, err_mop_illegal_cast is signaled.

os_type::operator os_class_type&()

operator os_class_type&();

Provides for safe casts from **os_type** to **os_class_type&**. If the cast is not permissible, err_mop_illegal_cast is signaled.

os_type::operator os_enum_type&()

operator os_enum_type&();

Provides for safe casts from **os_type** to **os_enum_type&**. If the cast is not permissible, err_mop_illegal_cast is signaled.

os_type::operator os_function_type&()

operator os_function_type&();

Provides for safe casts from **os_type** to **os_function_type&**. If the cast is not permissible, err_mop_illegal_cast is signaled.

os_type::operator os_instantiated_class_type&()

operator os_instantiated_class_type&();

Provides for safe casts from **os_type** to **os_instantiated_class_type&**. If the cast is not permissible, err_mop_illegal_cast is signaled.

os_type::operator os_integral_type&()

operator os_integral_type&();

Provides for safe casts from **os_type** to **os_integral_type&**. If the cast is not permissible, err_mop_illegal_cast is signaled.

os_type::operator os_named_indirect_type&()

operator os_named_indirect_type&();

Provides for safe casts from **os_type** to **os_named_indirect_type&**. If the cast is not permissible, err_mop_illegal_cast is signaled.

os_type::operator os_pointer_type&()

operator os_pointer_type&();

Provides for safe casts from **os_type** to **os_pointer_type&**. If the cast is not permissible, err_mop_illegal_cast is signaled.

os_type::operator os_pointer_to_member_type&()

operator os_pointer_to_member_type&();

Provides for safe casts from **os_type** to **os_pointer_to_member_type&**. If the cast is not permissible, err_mop_illegal_cast is signaled.

os_type::operator os_real_type&()

operator os_real_type&();

Provides for safe casts from **os_type** to **os_real_type&**. If the cast is not permissible, err_mop_illegal_cast is signaled.

os_type::operator os_reference_type&()

operator os_reference_type&();

Provides for safe casts from **os_type** to **os_reference_type&**. If the cast is not permissible, err_mop_illegal_cast is signaled.

os_type::operator os_type_type&()

operator os_type_type&();

Provides for safe casts from **os_type** to **os_type_type&**. If the cast is not permissible, err_mop_illegal_cast is signaled.

os_type::operator os_void_type&()

operator os_void_type&();

Provides for safe casts from **os_type** to **os_void_type&**. If the cast is not permissible, err_mop_illegal_cast is signaled.

os_type::Pointer

This enumerator indicates a pointer type. It is a possible return value from **os_type::get_kind()**.

os_type::Pointer_to_member

This enumerator indicates a pointer-to-member type. It is a possible return value from **os_type::get_kind()**.

os_type::Reference

This enumerator indicates a reference type. It is a possible return value from **os_type::get_kind()**.

os_type::set_alignment()

void set_alignment(os_unsigned_int32);

Sets the alignment associated with the type.

os_type::set_size()

void set_size (os_unsigned_int32 size) _OS_throw (err_mop);

Sets the size of the specified type in bytes.

os_type::Signed_char

This enumerator indicates the type **signed char**. It is a possible return value from **os_type::get_kind()**.

os_type::Signed_long

This enumerator indicates the type **long**. It is a possible return value from **os_type::get_kind()**.

os_type::Signed_short

This enumerator indicates the type **short**. It is a possible return value from **os_type::get_kind()**.

os_type::strip_indirect_types()

const os_type &strip_indirect_types() const;

For types with **const** or **volatile** specifiers, this function returns a **const** reference to the type being specified as **const** or **volatile**. For example, if the specified **os_type** represents the type **const int**,

strip_indirect_types() will return an **os_type** representing the type **int**. If the specified **os_type** represents the type **char const * const**, **strip_indirect_types()** will return an **os_type** representing the type **char const ***.

For typedefs, this function returns the original type for which the typedef is an alias.

This function calls itself recursively until the result is not an **os_ indirect_type**. So, for example, consider an **os_named_indirect_ type** representing

typedef const part const_part

The result of applying **strip_indirect_types()** to this is an **os_class_ type** representing the class **part** (*not* an **os_anonymous_indirect_ type** representing **const part** — which would be the result of **os_ indirect_type::get_target_type()**).

os_type &strip_indirect_types();

For types with **const** or **volatile** specifiers, this function returns a non-**const** reference to the type being specified as **const** or **volatile**. For example, if the specified **os_type** represents the type **const int**, **strip_indirect_types()** will return an **os_type** representing the type **int**. If the specified **os_type** represents the type **char const * const**, **strip_indirect_types()** will return an **os_type** representing the type **char const ***.

For typedefs, this function returns the original type for which the typedef is an alias.

This function calls itself recursively until the result is not an **os_ indirect_type**. So, for example, consider an **os_named_indirect_ type** representing

typedef const part const_part

The result of applying **strip_indirect_types()** to this is an **os_class_ type** representing the class **part** (*not* an **os_anonymous_indirect_ type** representing **const part** — which would be the result of **os_ indirect_type::get_target_type()**).

os_type::Type

This enumerator indicates a type serving as a parameter to a class or function template. It is a possible return value from **os_type::get_kind()**.

os_type::type_at()

```
static const os_type *type_at(const void *p);
```

Returns the type of the object that begins at location **p**. **p** should point to the beginning of an instance of a class or built-in type such as **int**. The instance can be allocated at top level, or it can be the value of a data member or the subobject corresponding to a base class. If **p** does not point to the beginning of such an instance, **0** is returned. If two or more objects start at **p**, the type of the innermost object is returned.

os_type::type_containing()

```
static const os_type *type_containing
    const void *p,
    const void *&p_container,
    os_unsigned_int32 &element_count
);
```

If the outermost object that contains **the location p** is not an array, this function returns the type of the outermost containing object. The argument **p_container** is modified to point to the containing object, and **element_count** is set to **1**.

If the outermost containing object is an array, the function sets **element_count** to the number of elements in the array, adjusts **p_container** to point to the first element of the array, and returns the element type of the array.

If **p** points to freed or transient memory, **0** is returned, and **p_container** and **element_count** are set to **0**.

os_type::Undefined

This enumerator indicates a type whose kind cannot be determined. It is a possible return value from **os_type::get_kind()**.

os_type::Unsigned_char

This enumerator indicates the type **unsigned char**. It is a possible return value from **os_type::get_kind()**.

os_type::Unsigned_integer

This enumerator indicates the type **unsigned int**. It is a possible return value from **os_type::get_kind()**.

os_type::Unsigned_long

This enumerator indicates the type **unsigned long**. It is a possible return value from **os_type::get_kind()**.

os_type::Unsigned_short

This enumerator indicates the type **unsigned short**. It is a possible return value from **os_type::get_kind()**.

os_type::Void

This enumerator indicates the type **void**. It is a possible return value from **os_type::get_kind()**.

os_type_template

class os_type_template : public os_template

This class is part of the ObjectStore metaobject protocol, which provides access to ObjectStore schemas. Instances of this class represent parameterized types. **os_type_template** is derived from **os_template**.

os_type_template::create()

```
static os_type_template& create(
    os_type*,
    os_List<os_template_formal_arg*>&
);
```

Creates a template that is a parameterization of the specified type with the specified parameters.

os_type_template::get_type()

const os_type &get_type() const;

Returns the type being parameterized.

os_type_template::set_type()

void set_type(os_type&);

Specifies the type being parameterized.

os_type_template_actual_arg

class os_type_template_actual_arg : public os_template_actual_arg

This class is part of the ObjectStore metaobject protocol, which provides access to ObjectStore schemas. Instances of this class represent types that are actual parameters of class templates.

os_type_template_actual_arg::create()

static os_type_template_actual_arg& create(os_type*);

Creates an actual parameter consisting of the specified type.

os_type_template_actual_arg::get_type()

const os_type &get_type() const;

Returns a reference to a **const** type, the type of which the actual parameter consists.

os_type &get_type();

Returns a reference to a non-**const** type, the type of which the actual parameter consists.

os_type_template_actual_arg::set_type()

void set_type(os_type&);

Sets the type of which the actual parameter consists.

os_type_type

class os_type_type : public os_type

This class is part of the ObjectStore metaobject protocol, which provides access to ObjectStore schemas. An instance of this class represents a type serving as a parameter to a class or function template. This class is derived from **os_type**. The member functions **os_type::get_size()** and **os_type::get_alignment()** signal err_mop when invoked on an **os_type_type**.

os_typed_pointer_void

Instances of this class encapsulate a **void*** pointer and an object representing the type of the object pointed to. Instances of **os_typed_pointer_void** are returned by **os_schema_evolution::get_evolved_address()**, **os_schema_evolution::get_unevolved_address()**, **os_schema_evolution::get_evolved_object()**, and **os_schema_evolution::get_unevolved_object()**.

os_typed_pointer_void::get_type()

const os_type &get_type() const;

Returns a reference to an **os_type** representing the class of the object pointed to by the specified **os_typed_pointer_void**.

os_typed_pointer_void::operator void*()

operator void*() const;

Returns the **void*** pointer encapsulated by the specified **os_typed_pointer_void**.

os_typespec

Instances of this class are passed to persistent **new** by applications using the ObjectStore C++ library interface. Typespecs must be transiently allocated; you should not allocate a typespec in persistent memory.

If a user-defined class includes a member declaration exactly like this:

static os_typespec *get_os_typespec();

the ObjectStore schema generator will automatically supply a body for this function, which will return a pointer to a typespec for the class. The first time such a function is called by a particular process, it will allocate the typespec and return a pointer to it. Subsequent calls to the function in the same process do not result in further allocation; instead, a pointer to the same **os_typespec** object is returned.

All ObjectStore programs must include the header file **<ostore/ostore.hh>**.

os_typespec::get_char()

static os_typespec *get_char();

Returns a pointer to a typespec for the type **char**. The first time such a function is called by a particular process, it will allocate the typespec and return a pointer to it. Subsequent calls to the function in the same process do not result in further allocation; instead, a pointer to the same **os_typespec** object is returned.

os_typespec::get_double()

static os_typespec *get_double();

Returns a pointer to a typespec for the type **double**. The first time such a function is called by a particular process, it will allocate the typespec and return a pointer to it. Subsequent calls to the function in the same process do not result in further allocation; instead, a pointer to the same **os_typespec** object is returned.

os_typespec::get_float()

static os_typespec *get_float();

Returns a pointer to a typespec for the type **float**. The first time such a function is called by a particular process, it will allocate the typespec and return a pointer to it. Subsequent calls to the function in the same process do not result in further allocation; instead, a pointer to the same **os_typespec** object is returned.

os_typespec::get_int()

static os_typespec *get_int();

Returns a pointer to a typespec for the type **int**. The first time such a function is called by a particular process, it will allocate the typespec and return a pointer to it. Subsequent calls to the function in the same process do not result in further allocation; instead, a pointer to the same **os_typespec** object is returned.

os_typespec::get_long()

static os_typespec *get_long();

Returns a pointer to a typespec for the type **long**. The first time such a function is called by a particular process, it will allocate the typespec and return a pointer to it. Subsequent calls to the function in the same process do not result in further allocation; instead, a pointer to the same **os_typespec** object is returned.

os_typespec::get_long_double()

static os_typespec *get_long_double();

Returns a pointer to a typespec for the type **long_double**. The first time such a function is called by a particular process, it will allocate the typespec and return a pointer to it. Subsequent calls to the function in the same process do not result in further allocation; instead, a pointer to the same **os_typespec** object is returned.

os_typespec::get_name()

char *get_name();

Returns the name of the type designated by the specified typespec.

os_typespec::get_pointer()

static os_typespec *get_pointer();

Returns a pointer to a typespec for the type **void***. The first time such a function is called by a particular process, it will allocate the typespec and return a pointer to it. Subsequent calls to the function in the same process do not result in further allocation; instead, a pointer to the same **os_typespec** object is returned.

os_typespec::get_short()

static os_typespec *get_short();

Returns a pointer to a typespec for the type **short**. The first time such a function is called by a particular process, it will allocate the typespec and return a pointer to it. Subsequent calls to the function in the same process do not result in further allocation; instead, a pointer to the same **os_typespec** object is returned.

os_typespec::get_signed_char()

static os_typespec *get_signed_char();

Returns a pointer to a typespec for the type **signed_char**. The first time such a function is called by a particular process, it will allocate the typespec and return a pointer to it. Subsequent calls to the function in the same process do not result in further allocation; instead, a pointer to the same **os_typespec** object is returned.

os_typespec::get_signed_int()

static os_typespec *get_signed_int();

Returns a pointer to a typespec for the type **signed_int**. The first time such a function is called by a particular process, it will allocate the typespec and return a pointer to it. Subsequent calls to the function in the same process do not result in further allocation; instead, a pointer to the same **os_typespec** object is returned.

os_typespec::get_signed_long()

static os_typespec *get_signed_long();

Returns a pointer to a typespec for the type **signed_long**. The first time such a function is called by a particular process, it will allocate the typespec and return a pointer to it. Subsequent calls to the function in the same process do not result in further allocation; instead, a pointer to the same **os_typespec** object is returned.

os_typespec::get_signed_short()

static os_typespec *get_signed_short();

Returns a pointer to a typespec for the type **signed_short**. The first time such a function is called by a particular process, it will allocate the typespec and return a pointer to it. Subsequent calls to the function in the same process do not result in further allocation; instead, a pointer to the same **os_typespec** object is returned.

os_typespec::get_unsigned_char()

static os_typespec *get_unsigned_char();

Returns a pointer to a typespec for the type **unsigned_char**. The first time such a function is called by a particular process, it will allocate the typespec and return a pointer to it. Subsequent calls to the function in the same process do not result in further allocation; instead, a pointer to the same **os_typespec** object is returned.

os_typespec::get_unsigned_int()

static os_typespec *get_unsigned_int();

Returns a pointer to a typespec for the type **unsigned_int**. The first time such a function is called by a particular process, it will allocate the typespec and return a pointer to it. Subsequent calls to the function in the same process do not result in further allocation; instead, a pointer to the same **os_typespec** object is returned.

os_typespec::get_unsigned_long()

static os_typespec *get_unsigned_long();

Returns a pointer to a typespec for the type **unsigned_long**. The first time such a function is called by a particular process, it will allocate the typespec and return a pointer to it. Subsequent calls to the function in the same process do not result in further allocation; instead, a pointer to the same **os_typespec** object is returned.

os_typespec::get_unsigned_short()

static os_typespec *get_unsigned_short();

Returns a pointer to a typespec for the type **unsigned_short**. The first time such a function is called by a particular process, it will allocate the typespec and return a pointer to it. Subsequent calls to the function in the same process do not result in further allocation; instead, a pointer to the same **os_typespec** object is returned.

os_typespec::operator ==()

os_boolean operator ==(const os_typespec&) const;

Returns nonzero (true) if the two typespecs designate the same type; returns **0** (false) otherwise.

os_typespec::os_typespec()

os_typespec(char *type_name);

Constructs an **os_typespec** representing the type with the specified name. This must be a name for a built-in type (such as **int** or **char**), a class, or a pointer-to-class or pointer-to-built-in type. For example, **foo**** is not allowed — use **void*** instead. Typedef names are not allowed. Typespecs must be allocated transiently; they should not be allocated in persistent memory.

In general, make use of typespecs for fundamental types, described in the *ObjectStore C++ API User Guide* in Typespecs for Fundamental Types, or typespecs for classes, described in Typespecs for Classes, rather than the **os_typespec** constructor. If you must use the **os_typespec** constructor, be sure to adhere to the guidelines outlined in The os_typespec Constructor in the *ObjectStore C++ API User Guide*.

If you use **const** or **volatile** in specifying the type, always put the **const** or **volatile** specifier after the type it modifies. For example, use

char const *

rather than

const char *

os_void_type

class os_void_type : public os_type

This class is part of the ObjectStore metaobject protocol, which provides access to ObjectStore schemas. An instance of this class represents the type **void**. **os_void_type** is derived from **os_type**. The member functions **os_type::get_size()** and **os_type::get_ alignment()** signal err_mop when invoked on an **os_void_type**.

Programs using this class must include **<ostore/ostore.hh>**, followed by **<ostore/coll.hh>** (if used), followed by **<ostore/mop.hh>**.

os_void_type::create()

static os_void_type& create();

Creates an object representing the type **void**.

os_with_mapped

os_with_mapped::os_with_mapped()

```
os_with_mapped (
        void *location,
        os_unsigned_int32 size,
        os_boolean for_write = 0
        );
```

Creates an **os_with_mapped** object that ensures that the persistent range indicated in the **location** and **size** parameters is accessible to system calls and library functions, by forcing the range to stay in the cache and accessible in memory.

location specifies the starting address of the range

size specifies the size of the range in bytes

for_write is a Boolean value that indicates whether you intend to update the object.

Note that an **os_mapped** object cannot be used across transaction boundaries, including nested transactions.

See Ensuring Data Access During System Calls in *ObjectStore C++ API User Guide* for an example.

os_with_mapped::~os_with_mapped()

```
~os_with_mapped();
```

Restores normal cache behavior with respect to the range of data specified at the time the **os_with_mapped** object was created.

Chapter 3
System-Supplied Global Functions

Some system-supplied interface functions are not members of any class, but are global C++ functions. These are listed alphabetically in this chapter.

Functions

::operator delete()	366
::operator new()	367
::os_fetch()	370
::os_fetch_address()	373
::os_store()	374

::operator delete()

Persistent storage can be reclaimed using the **delete** operator, just as it would be used for transient objects.

When a persistent object is deleted with **operator delete()**, the database in which it is to be stored must be opened by the process performing the deletion, and this process must have a transaction in progress.

This function can also be used to reclaim transient storage. See **objectstore::set_transient_delete_function()** on page 41.

On the OS/2 operating system, ObjectStore also supports **::operator delete[]()** for persistent storage, which works just like **::operator delete()**.

Required header files

All ObjectStore programs must include the header file **<ostore/ostore.hh>**.

void ::operator delete()

Deletes the specified object from storage. The argument must point to the beginning of an object's storage. If it points to the middle of an object, the exception err_invalid_deletion is signaled. This might happen, for example, if the argument points to the fifth element of a persistent array, or if the argument has been cast from a **D*** to a **B***, where class **D** is derived directly from class **B** and some other class.

::operator new()

For creation of persistently allocated objects, there are several system-supplied overloadings of the C++ global function **operator new()**. These functions take one argument that specifies clustering information, one argument that specifies the type of the new object, and, for arrays, an argument specifying the number of elements.

The clustering information indicates the database in which the new object is to be stored, and can also specify the particular segment or object cluster within that database in which the object is to reside. Specifying the transient database or segment results in transient allocation.

The argument indicating the type of the new object is an instance of the class **os_typespec**.

Each function returns a pointer to newly allocated memory. Persistent memory is initialized to null by ObjectStore.

ObjectStore supports cross-database pointers. You can allow cross-database pointers from a given database or segment with **os_database::allow_external_pointers()** or **os_segment::allow_ external_pointers()**.

If cross-database pointers are not enabled for a database or segment, and memory allocated within it is assigned pointers to memory allocated in a different database, the pointers are valid only until the end of the outermost transaction in which the assignment occurred.

ObjectStore also supports cross-transaction pointers. You can enable cross-transaction pointers at any point in the execution of an application with **objectstore::retain_persistent_addresses()**.

If cross-transaction pointers are not enabled, a pointer from transiently allocated memory to persistently allocated memory is valid only until the end of the outermost transaction in which the pointer was assigned to that transient memory. Attempts to dereference an invalid pointer in subsequent transactions might access arbitrary data, unless data from the segment was committed only with the segment in **check_illegal_pointers** mode. See **os_segment::check_illegal_pointers()**.

If persistent memory is assigned pointers to transiently allocated memory, the pointers are valid only until the end of the outermost transaction in which the assignment occurred.

Do not pass pointers to persistent memory to non-ObjectStore processes using system calls or interprocess communication.

Before a persistent object can be created with **operator new()**, the database in which it is to be stored must exist. Moreover, this database must be opened by the process performing the creation, and this process must have a transaction in progress.

The type **os_int32**, mentioned below, is defined as a signed 32-bit integer type.

Required files

All ObjectStore programs must include the header file **<ostore/ostore.hh>** and call **objectstore::initialize()**.

void*::operator new() overloadings

void *::operator new(size_t, os_database*, os_typespec*)

If this form is used, a newly created object of the type specified by the **os_typespec** is stored in the specified database. If the transient database (see **os_database::get_transient_database()** on page 86) is specified, the new object is transiently allocated. If you specify the transient database, you can supply **0** for the **os_typespec*** argument.

void *::operator new(size_t, os_database*, os_typespec*, os_int32)

If this form is used, a newly created array of objects of the type specified by the **os_typespec** is stored in the specified database. If the transient database (see **os_segment::get_transient_database()**) is specified, the new array is transiently allocated. If you specify the transient database, you can supply **0** for the **os_typespec*** argument. The **os_int32** specifies the number of elements in the new array. On OS/2, an overloading of **::operator new[]()** is provided instead. It has the same arguments and return type.

void *::operator new(size_t, os_segment*, os_typespec*)

If this form is used, a newly created object of the type specified by the **os_typespec** is stored in the specified segment (unless that segment is the schema segment, in which case err_misc is signaled). If the transient segment (see **os_database::get_ transient_segment()**) is specified, the new object is transiently allocated. If you specify the transient segment, you can supply **0** for the **os_typespec*** argument.

void *::operator new(size_t, os_segment*, os_typespec*, os_int32)

If this form is used, a newly created array of objects of the type specified by the **os_typespec** is stored in the specified segment (unless that segment is the schema segment, in which case err_misc is signaled). If the transient segment (see **os_segment::get_ transient_segment()** on page 299) is specified, the new array is transiently allocated. If you specify the transient segment, you can supply **0** for the **os_typespec*** argument. The **os_int32** specifies the number of elements in the new array. On OS/2, an overloading of **::operator new[]()** is provided instead. It has the same arguments and return type.

void *::operator new(size_t, os_object_cluster*, os_typespec*)

If this form is used, a newly created object of the type specified by the **os_typespec** is stored in the specified object cluster, unless the new object does not fit in the cluster, in which case err_cluster_full is signaled.

void *::operator new(size_t, os_object_cluster*, os_typespec*, os_ int32)

If this form is used, a newly created array of objects of the type specified by the **os_typespec** is stored in the specified object cluster, unless the new array does not fit in the cluster, in which case err_cluster_full is signaled. The **os_int32** specifies the number of elements in the new array. On OS/2, an overloading of **::operator new[]()** is provided instead. It has the same arguments and return type.

Note: The overloadings of **operator new()** described in this section can be used for transient as well as persistent allocation. To allocate transient memory, supply a pointer to the transient database or segment for the **os_database*** or **os_segment*** argument. If you specify the transient database or segment, you can supply **0** for the **os_typespec*** argument.

Performing **os_database::of()** or **os_segment::of()** on a pointer to transient memory returns a pointer to the transient database or segment, respectively. The following static member functions also return a pointer to the transient database or segment: **os_ database::get_transient_database()** and **os_segment::get_ transient_segment()**.

::os_fetch()

Retrieves the value of a specified data member for a specified object.

void *os_fetch

```
void *os_fetch (
    const void* obj,
    const os_member_variable& mem,
    void *&val
);
```

Retrieves the value of the member represented by **mem** for the object **obj** by setting **val** to a reference to the value. err_mop is signaled if the value type of the specified member is not compatible with **val**.

unsigned long os_fetch

```
unsigned long os_fetch (
    const void* obj,
    const os_member_variable& mem,
    unsigned long &val
);
```

Retrieves the value of the member represented by **mem** for the object **obj** by setting **val** to a reference to the value. err_mop is signaled if the value type of the specified member is not compatible with **val**.

long os_fetch

```
long os_fetch (
    const void* obj,
    const os_member_variable& mem,
    long &val
);
```

Retrieves the value of the member represented by **mem** for the object **obj** by setting **val** to a reference to the value. err_mop is signaled if the value type of the specified member is not compatible with **val**.

unsigned int os_fetch

```
unsigned int os_fetch (
    const void* obj,
    const os_member_variable& mem,
    unsigned int &val
);
```

Retrieves the value of the member represented by **mem** for the object **obj** by setting **val** to a reference to the value. err_mop is signaled if the value type of the specified member is not compatible with **val**.

int os_fetch

```
int os_fetch (
    const void* obj,
    const os_member_variable& mem,
    int &val
);
```

Retrieves the value of the member represented by **mem** for the object **obj** by setting **val** to a reference to the value. err_mop is signaled if the value type of the specified member is not compatible with **val**.

unsigned short os_fetch

```
unsigned short os_fetch (
    const void* obj,
    const os_member_variable& mem,
    unsigned short &val
);
```

Retrieves the value of the member represented by **mem** for the object **obj** by setting **val** to a reference to the value. err_mop is signaled if the value type of the specified member is not compatible with **val**.

short os_fetch

```
short os_fetch (
    const void* obj,
    const os_member_variable& mem,
    short &val
);
```

Retrieves the value of the member represented by **mem** for the object **obj** by setting **val** to a reference to the value. err_mop is signaled if the value type of the specified member is not compatible with **val**.

unsigned char os_fetch

```
unsigned char os_fetch (
    const void* obj,
    const os_member_variable& mem,
    unsigned char &val
);
```

Retrieves the value of the member represented by **mem** for the object **obj** by setting **val** to a reference to the value. err_mop is signaled if the value type of the specified member is not compatible with **val**.

char os_fetch

```
char os_fetch (
    const void* obj,
    const os_member_variable& mem,
    char &val
);
```

Retrieves the value of the member represented by **mem** for the object **obj** by setting **val** to a reference to the value. err_mop is signaled if the value type of the specified member is not compatible with **val**.

float os_fetch

```
float os_fetch (
    const void* obj,
    const os_member_variable& mem,
    float &val
);
```

Retrieves the value of the member represented by **mem** for the object **obj** by setting **val** to a reference to the value. err_mop is signaled if the value type of the specified member is not compatible with **val**.

double os_fetch

```
double os_fetch (
    const void* obj,
    const os_member_variable& mem,
    double &val
);
```

Retrieves the value of the member represented by **mem** for the object **obj** by setting **val** to a reference to the value. err_mop is signaled if the value type of the specified member is not compatible with **val**.

long double os_fetch

```
long double os_fetch (
    const void* obj,
    const os_member_variable& mem,
    long double &val
);
```

Retrieves the value of the member represented by **mem** for the object **obj** by setting **val** to a reference to the value. err_mop is signaled if the value type of the specified member is not compatible with **val**.

::os_fetch_address()

Retrieves the address of a specified data member for a given object, or the address of the subobject corresponding to a specified base class for a given object.

void *os_fetch_ address

```
void *os_fetch_address (
    void *obj,
    const os_member_variable& mem
);
```

Retrieves the address of the data member **mem** for the object pointed to by **obj**. err_mop is signaled if **mem** is an **os_field_ member_variable**.

void *os_fetch_ address

```
void *os_fetch_address (
    void *obj,
    const os_base_class &b
);
```

Retrieves the address of the subobject corresponding to base class **b** for the object pointed to by **obj**.

::os_store()

Stores the value of a specified data member for a specified object. There are several overloadings:

void os_store

```
void os_store (
    void* obj,
    const os_member_variable& mem,
    const void *val
);
```

Establishes **val** as the value of the member represented by **mem** for the object **obj**. err_mop is signaled if the value type of the specified member is not compatible with **val**.

void os_store

```
void os_store (
    void* obj,
    const os_member_variable& mem,
    const unsigned long val
);
```

Establishes **val** as the value of the member represented by **mem** for the object **obj**. err_mop is signaled if the value type of the specified member is not compatible with **val**.

void os_store

```
void os_store (
    void* obj,
    const os_member_variable& mem,
    const long val
);
```

Establishes **val** as the value of the member represented by **mem** for the object **obj**. err_mop is signaled if the value type of the specified member is not compatible with **val**.

void os_store

```
void os_store (
    void* obj,
    const os_member_variable& mem,
    const unsigned int val
);
```

Establishes **val** as the value of the member represented by **mem** for the object **obj**. err_mop is signaled if the value type of the specified member is not compatible with **val**.

void os_store

```
void os_store (
    void* obj,
    const os_member_variable& mem,
    const int val
);
```

Establishes **val** as the value of the member represented by **mem** for the object **obj**. err_mop is signaled if the value type of the specified member is not compatible with **val**.

void os_store

```
void os_store (
    void* obj,
    const os_member_variable& mem,
    const unsigned short val
);
```

Establishes **val** as the value of the member represented by **mem** for the object **obj**. err_mop is signaled if the value type of the specified member is not compatible with **val**.

void os_store

```
void os_store (
    void* obj,
    const os_member_variable& mem,
    const short val
);
```

Establishes **val** as the value of the member represented by **mem** for the object **obj**. err_mop is signaled if the value type of the specified member is not compatible with **val**.

void os_store

```
void os_store (
    void* obj,
    const os_member_variable& mem,
    const unsigned char val
);
```

Establishes **val** as the value of the member represented by **mem** for the object **obj**. err_mop is signaled if the value type of the specified member is not compatible with **val**.

void os_store

```
void os_store (
    void* obj,
    const os_member_variable& mem,
    const char val
);
```

Establishes **val** as the value of the member represented by **mem** for the object **obj**. err_mop is signaled if the value type of the specified member is not compatible with **val**.

void os_store

```
void os_store (
    void* obj,
    const os_member_variable& mem,
    const float val
);
```

Establishes **val** as the value of the member represented by **mem** for the object **obj**. err_mop is signaled if the value type of the specified member is not compatible with **val**.

void os_store

```
void os_store (
    void* obj,
    const os_member_variable& mem,
    const double val
);
```

Establishes **val** as the value of the member represented by **mem** for the object **obj**. err_mop is signaled if the value type of the specified member is not compatible with **val**.

void os_store

```
void os_store (
    void* obj,
    const os_member_variable& mem,
    const long double val
);
```

Establishes **val** as the value of the member represented by **mem** for the object **obj**. err_mop is signaled if the value type of the specified member is not compatible with **val**.

Chapter 4
System-Supplied Macros

This chapter describes the ObjectStore macros related to transactions and schema. See the *ObjectStore Collections C++ API Reference* for information on macros related to index maintenance.

Macros

OS_BEGIN_TXN(), OS_BEGIN_TXN_NAMED(), and OS_END_TXN() 378

OS_ESTABLISH_FAULT_HANDLER and OS_END_FAULT_HANDLER 383

OS_MARK_QUERY_FUNCTION() 385

OS_MARK_SCHEMA_TYPE() OS_MARK_SCHEMA_TYPESPEC() 386

Component Schema Macros

OS_BEGIN_TXN(), OS_BEGIN_TXN_NAMED(), and OS_END_TXN()

For applications using only the ObjectStore C++ library interface, transaction macros are supplied. **OS_BEGIN_TXN()** begins a transaction and establishes a new scope; **OS_END_TXN()** ends and commits the transaction started by the previous matching **OS_BEGIN_TXN()** call. **OS_END_TXN()** also marks the end of the scope established by **OS_BEGIN_TXN()**.

OS_BEGIN_TXN_NAMED() is identical in behavior to **OS_BEGIN_TXN()** except that it allows you to specify a transaction name. The name must be unique among transactions in the same function. It is used in constructing statement labels.

Synopsis of Transaction Macros

Form of the call

OS_BEGIN_TXN (*txn-tag***,***expression***,***txn-type***)**
...
OS_END_TXN (*txn-tag***)**

or

OS_BEGIN_TXN_NAMED (*txn-tag***,***expression***,***txn-type***,***txn-name***)**
...
OS_END_TXN (*txn-tag***)**

txn-tag is an identifier, a tag for the transaction that is not used for any other transaction in the same function. (The tags are used to construct statement labels, and so have the same scope as labels in C++.) The tag specified in a transaction's **BEGIN** and **END** must be the same.

expression is of type **tix_exception****, and specifies a location in which ObjectStore stores the transaction's completion status. If the **tix_exception**** points to **0** when the transaction completes, the transaction committed (that is, its changes were made permanent and visible). If the **tix_exception**** points to a **tix_exception***, the transaction was aborted (that is, its changes to persistent state were undone), and the identity of the exception indicates the nature of the abort.

txn-type is one of

os_transaction::abort_only

os_transaction::read_only

os_transaction::update

os_transaction::abort_only indicates a transaction that can write to persistent data, but cannot be committed. See **os_ transaction::abort_only** in *ObjectStore C++ API Reference* for further information.

os_transaction::read_only indicates that the transaction performs no updates to persistent storage. If write access to persistent data is attempted, a run-time error is signaled.

os_transaction::update indicates that the transaction performs updates to persistent storage.

txn-name is the name you want to give to the transaction. See **os_ transaction::get_name()** on page 336.

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.

Transactions can be nested, but a nested transaction must be of the same type (**os_transaction::update** or **os_transaction::read_only**) as its parent, otherwise err_trans_wrong_type is signaled.

Locking

Locking in ObjectStore works as follows. When a persistent data item is read, the page on which it resides is read-locked by ObjectStore. When a data item is written, the page on which it resides is write-locked by the system. When another process attempts write access to an item on a read-locked page, the access is delayed until the outermost transaction (within which the access occurs) completes and the page is unlocked. Read access to a read-locked page is not delayed. When another process attempts read or write access to an item on a write-locked page, the access is delayed until the outermost transaction (in which the access occurs) completes and the page is unlocked.

Note that for purposes of locking, the boundaries of a nested transaction are insignificant. Locks acquired during a nested transaction are not released until the nonnested transaction within which the lock is nested terminates. However, when a nested transaction aborts, its changes to the database are effectively erased, and the state of the database prior to the initiation of the nested transaction is restored.

Transaction Aborts

When a transaction aborts, its changes to the database are undone, and the database is effectively restored to its state as of the moment the transaction began. In addition, if it is not nested, the transaction releases all its locks.

There are two kinds of aborts: user aborts and system aborts.

User aborts

The user can specify that an abort occur at a certain point in the flow of control by invoking the member function **os_ transaction::abort()** or **os_transaction::abort_top_level()**.

System aborts

On rare occasions, the system aborts a transaction either because of a network failure or because it has determined that the transaction is involved in a *deadlock*. A simple deadlock occurs when one transaction holds a lock on a page that another transaction is waiting to access, while at the same time this other transaction holds a lock on a page that the first transaction is waiting to access. Each process cannot proceed until the other does, with the result that neither process can proceed. Other forms of deadlock are analogous.

When a system abort occurs as a result of a deadlock, the transaction is automatically retried (after undo handling) until it completes successfully or until the maximum number of retries has occurred. This maximum for any transaction in a given process is determined by the value of the private static data member **os_transaction::max_retries**. Its default value is **10**. You can change or retrieve the value of **os_transaction::max_retries** with **os_transaction::set_max_retries()** and **os_transaction::get_ max_retries()**. Changing **max_retries** remains in effect only for the duration of the process, and is invisible to other processes. (Note that this does not apply to dynamic transactions, which are not automatically retried.)

Two-Phase Commit

Two-phase commit is the process by which transactions that use multiple Servers commit in a consistent manner. The majority of transactions, which do *not* use multiple Servers, are unaffected by two-phase commit.

Transactions that write-access data on multiple Servers commit in two phases: a voting phase and a decision phase. First, all the Servers vote on a transaction, indicating whether they are able to commit it or not. Servers always vote yes unless they have crashed fatally, such as by running out of disk space. If all the votes are yes, the decision is made to commit, and all the Servers are told to commit. (Note that, during normal processing, the Client process is the coordinator of the transaction.)

Problems can still occur at this point, before the transaction completes: a Server might crash in the middle of the commit process — after it has voted, but before it receives a decision. Another possibility is that a Server could crash or be restarted between the vote and decision phases. In both these cases, the Server must communicate with another Server to determine the outcome of the transaction, and the coordination control is passed to one of the Servers.

When a Server is restarted (perhaps after a power failure) after a multi-Server transaction was in the middle of committing when the crash occurred, it prints a message such as the following into the log file, or on standard output:

```
recovaux.C:1123(-2): This server is a participant in a multi server
transaction
recovaux.C:1127(-2): (id: 8 global id: 67372226,649525822,20971521)
recovaux.C:1138(-2): The coordinator of the transaction is:
recovaux.C:1143(-2): peachbottom
recovaux.C:1147(-2): There are 2 blocks involved in the transaction on
this server.
Server started
```

This indicates that during the restart process, the Server noticed that a multi-Server transaction was in the process of committing when the host system crashed. The information about transaction IDs can be used to match up outstanding transactions with other Servers that might also be recovering. The message about the coordinator indicates that it will receive the decision from that host (in this case, peachbottom) when that Server recovers.

During peachbottom's restart process, it too notices that a multi-Server transaction was in the commit process when the host crashed, and prints a similar message:

```
recovaux.C:1120(-2): This server is coordinator for a multi server
transaction
recovaux.C:1127(-2): (id: 6 global id: 67372226,649525822,20971521)
recovaux.C:1130(-2): The transaction committed.
recovaux.C:1135(-2): The 2 participant servers in the transaction are:
recovaux.C:1143(-2): peachbottom
recovaux.C:1143(-2): seabrook
recovaux.C:1147(-2): There are 0 blocks involved in the transaction on
this server.
Server started
```

(Note that it identifies the participant Servers, including itself.)

peachbottom then proceeds to notify seabrook of the decision for the transaction (committed), at which time seabrook prints the message

```
recovaux.C:1234(-2): Transaction committed
recovaux.C:1252(-2): (id=8 global id: 67372226,649525822,20971521)
```

The recovery is now complete for this transaction.

The pages that are involved in this transaction are in limbo during the commit phases as well as during the restart processing. However, during two-phase commit restart processing, it is possible for other requests to the Servers to be processed once they are started up. If one of those requests is for one of the pages that is being recovered as part of a multi-Server transaction that was committing, the client program receives an error indicating that the block is not recovered. This error is defined as

```
DEFINE_EXCEPTION(err_svr_blknotrecovered,
    "The block is blocked by an incomplete two phase commit",
    &err_svr);
```

Once the recovery processing is done for the corresponding two-phase commit, the block becomes free to use again.

OS_ESTABLISH_FAULT_HANDLER and OS_END_FAULT_HANDLER

ObjectStore must handle all memory access violations, because some are actually references to persistent memory in an ObjectStore database. On UNIX systems, ObjectStore can register a signal handler for such access violations. On Windows NT, there is no function for registering such a handler that will also work when you are debugging an ObjectStore application; the **SetUnhandledExceptionFilter** function does not work when you are using the Visual C++ debugger.

WIN

Therefore, every Windows NT ObjectStore application must put a handler for access violation at the top of every stack in the program. This normally means putting a handler in a program's **main** or **WinMain** function and, if the program uses multiple threads, putting a handler in the first function of new threads.

Use ObjectStore Macros to Set Up the Exception Handler

ObjectStore provides two macros to use in establishing a fault handler:

- **OS_ESTABLISH_FAULT_HANDLER**, which establishes the start of a fault handler block
- **OS_END_FAULT_HANDLER,** which ends the fault handler block

Using these macros, a typical **main** function in an ObjectStore application would look like

```
int main (int argc, char** argv) {
    OS_ESTABLISH_FAULT_HANDLER
        ...your code...
    OS_END_FAULT_HANDLER
    return value;
}
```

Braces and semicolons are not necessary, as they are part of the macros.

WIN

You must use these macros for ObjectStore applications on Windows NT, HP–UX, and Digital UNIX platforms. The reasons vary depending on platform. For example, on Windows NT, every stack in an ObjectStore program must be wrapped with the macros **OS_ESTABLISH_FAULT_HANDLER** and **OS_END_FAULT_**

HANDLER. This is because Windows NT offers no support for establishing a persistent memory fault handler in any other manner that would also work when you are debugging the application.

HPUX DIGU

On the HP–UX and Digital UNIX platforms, use the **OS_ ESTABLISH_FAULT_HANDLER** and **OS_END_FAULT_HANDLER** macros at the beginning and end of any thread that performs ObjectStore operations. This is needed because HP–UX and Digital UNIX signal handlers are installed strictly on a per-thread basis and are not inherited across **pthread_create** calls. See Establishing Fault Handlers in POSIX Thread Environments of *ObjectStore Building C++ Interface Applications* for additional information.

Note that these macros are only required for threads that use ObjectStore.

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, inside the dummy function containing the calls to **OS_MARK_SCHEMA_TYPE()**. No white space should appear in the argument list of **OS_MARK_QUERY_FUNCTION()**.

OS_MARK_SCHEMA_TYPE() OS_MARK_SCHEMA_TYPESPEC()

These macros are used to include a class in an application's schema.

Form of the calls

OS_MARK_SCHEMA_TYPE(*class*)

class is the class to be included in the application's schema. Typedef names are not allowed.

OS_MARK_SCHEMA_TYPESPEC((*type-name*<*x,y*>))

You can use this macro to parameterize types with multiple arguments. Note that you must place the type name and its arguments within a set of parentheses.

Calls to the macro should appear outside any function at global or file scope level.

Required include files

Schema source files must include the file **<ostore/manschem.hh>** after including **<ostore/ostore.hh>**, and should contain all the include lines from the application's source necessary to compile.

You must mark every class on which the application might perform persistent **new**, as well as every class whose instances can be used by the application as database entry points. You must also mark every class that appears in a library interface query string or index path in the application.

If you supply the **-make_reachable_source_classes_persistent** flag to the schema generator (see *ObjectStore Management*), you do not actually need to mark every class that might be read from a database. This flag causes every class that is both defined in the schema source file and reachable from a persistently marked class to be persistently allocatable and accessible. It is useful in ensuring that items that could be overlooked, such as subtypes, are marked; however, it can also make classes persistent that do not need to be in order for compilation to succeed.

Component Schema Macros

OS_REPORT_DLL_LOAD_AND_UNLOAD

Default: true

OS_REPORT_DLL_LOAD_AND_UNLOAD(*os_boolean***)**

Reports that a DLL has been loaded or unloaded.

Component Schema source file macro

When *os_boolean* is **true**, automatic reporting of DLL loading and unloading is enabled. This is accomplished by **ossg** generating code that calls **os_DLL_schema_info::DLL_loaded()** and **os_DLL_schema_info::DLL_unloaded()** from the DLL's initialization and termination functions.

If *os_boolean* is **false**, automatic reporting is not enabled and you must write your own code to call **os_DLL_schema_info::DLL_loaded()** and **os_DLL_schema_info::DLL_unloaded()** from the DLL's initialization and termination functions. The default is **true**.

OS_SCHEMA_DLL_ID

OS_SCHEMA_DLL_ID(*string***)**

Component Schema source file macro

For use in generating component schema, specifies the DLL identifier of the DLL. This macro can be used multiple times, for example, to specify different platform-specific DLL identifiers for different platforms. Do not conditionalize these calls on the platform– you want all the DLL identifiers to be recorded in any database that depends on this DLL.

You must call **OS_SCHEMA_DLL_ID** at least once in a DLL schema source file to distinguish it from an application schema.

OS_SCHEMA_INFO_NAME

OS_SCHEMA_INFO_NAME(*name***)**

Use with component schema source file

For use with component schema, generates a variable **extern os_DLL_schema_info** *name***;** that is the **os_DLL_schema_info** of this DLL. The default is to generate the schema information with a static name. Call this if you are going to call **os_DLL_schema_info::DLL_loaded()** yourself, so you can get access to the **os_DLL_schema_info**.

Use with application schema source file

This macro also works in an application schema, in which case the type of the variable is **os_application_schema_info** instead of **os_ DLL_schema_info**.

Chapter 5
User-Supplied Functions

Access to database services sometimes requires support from user-defined functions. The required functions are listed alphabetically in this chapter.

Function Discriminant Functions 390

Discriminant Functions

For each union-valued data member of each persistently allocated class, ObjectStore requires that the user provide an associated *discriminant function*, which indicates the field of the union currently in use. The function is used by the system at run time, when a union is brought into virtual memory from persistent storage. The discriminant function is a member function of the class defining the union-valued data member.

Similarly, for each union that is not a data member value, ObjectStore requires that the user provide a discriminant function. In this case, the discriminant function must be a member function of the union itself.

Each discriminant function is declared as follows:

os_int32 *function-name***();**

The type **os_int32** is defined as either **int** or **long**. The value returned is an integer indicating the union's active field (**1** for the first field, **2** for the second, and so on). A return value of **0** indicates an uninitialized union.

For a data member whose value is a vector of unions, a discriminant function of the same form is also required. Its return value indicates the active field of each of the vector's elements. (At a given time, all vector elements must have the same field as the active one.)

Note that a discriminant function cannot be virtual.

Each discriminant function's name serves to associate it with a union or vector of unions. The function name has the following form:

discriminant[*_union-name***[***_data-member-name***]]**

where *union-name* is the name of the associated union, and *data-member-name* is the name of the associated data member. The union name and data member name can be dropped if no ambiguity results. The data member name can be omitted when the class defining the discriminant function contains exactly one data member of the type. The union name can be omitted when the class defining the discriminant function contains exactly one union data member. The union name is actually redundant when the data member name is specified, but it is retained so that the discriminant function is self-documenting.

Discriminant functions are restricted with regard to the computations they can perform. Each discriminant member function of a given class must not reference data members other than those nonstatic members defined by that class and its nonvirtual bases. In addition, each such function must not rely on any global program state, perform any operations on pointers except comparison against 0, or invoke any virtual functions.

In addition, when accessing databases created by clients with a different byte ordering, access to a data member whose value occupies more than a single byte must be mediated by **objectstore::discriminant_swap_bytes()**. This function performs byte swapping that ObjectStore normally performs automatically. Use of this function is *only* required within discriminant functions; it is an error to invoke it in any other context. This function is declared as follows:

```
static void discriminant_swap_bytes(
    char *address, char *result, os_int32 n_bytes) ;
```

The **address** argument is the address of the member whose access is being mediated by this function. The **n_bytes** argument is the number of bytes to swap; possible values are 2, 4, and 8. The **result** argument points to memory allocated by the user to hold the correctly byte-swapped result; the allocated space should contain **n_bytes** bytes.

Below is a sample declaration of a class with a data member whose value is a union.

```
struct kgraph {
    typedef os_int32 tag_type
    char kg_type;
    union kg_union {
        struct kgraph *kg_kg;
        os_int32 kg_int;
    } kg;
    os_int32 discriminant(); // discriminant function for kg
};

os_int32 kgraph::discriminant()
{
    tag_type val;
    if (sizeof (tag_type)>1)
        // swap bytes for tags larger than char
        objectstore::discriminant_swap_bytes((char*)&kgtype,
            (char*)&val,
            sizeof(tag_type));
    else
        val = kgtype;
    return val;
}
```

If the value returned by the discriminant function is larger than the number of alternatives for the union, an exception such as the following is displayed:

```
No handler for exception:
Union discriminant function return value was out of bounds
<err-0025-0729>Union discriminant function returned a value larger than
the number of alternatives for data member kgraph.kg<0> of class
kgraph.
...
```

See Appendix A, Exception Facility, on page 555 for information about this exception.

At run time, when type determination is deemed necessary, the discriminant function is invoked. This function could have been named **discriminant_kg_union** or **discriminant_kg_union_kg**.

Chapter 6
C Library Interface

Although ObjectStore provides C++ class and function libraries, as well as a C++-based language binding, it is possible for plain C programs to access basic ObjectStore functionality. C functions and macros analogous to many of the functions provided in the C++ class and function libraries are listed in this chapter.

Topics

Overview	395
Getting Started	397
Building Blocks	398
objectstore Functions	402
os_database Functions	410
os_database_reference Functions	422
os_database_root Functions	423
os_dbutil Functions	424
os_keyword_arg Functions	433
os_keyword_arg_list Functions	436
os_lock_timeout_exception Functions	437
Metaobject Protocol Functions	439
os_notification Functions	500
os_object_cluster Functions	505
os_object_cursor Functions	507
os_pvar Functions	509
os_rawfs_entry Functions	510
os_segment Functions	513

os_segment_access Functions	520
os_server Functions	521
os_transaction Functions	523
os_typespec Functions	526
Reference Functions	529
Schema Evolution Functions	543
TIX Functions	552

Overview

ObjectStore includes a C library interface that allows access to many of ObjectStore's features directly from C programs. These features include

- Databases and segments
- Roots
- Transactions
- References
- Metaobject protocol
- Notifications
- Schema evolution

For information on the C interface for collections and queries, see *ObjectStore Collections 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 A, Exception Facility, on page 555.

If you are using 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>
}
```

If you are using collections, follow this with

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

```
extern "C" {
#include <ostore/coll.h>
}
```

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.

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.

Building Blocks

objectstore_initialize()

extern void objectstore_initialize();

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

Persistent Allocation and Type Specifiers

You allocate persistent memory with the following functions:

- **os_database_alloc()**
- **os_segment_alloc()**
- **os_object_cluster_alloc()**

Each of these functions requires an **os_typespec** as one of its arguments. Functions for creating and manipulating **os_ typespec**s are also described in this section. Persistent storage can be reclaimed using the function **objectstore_delete()**, which is described here as well.

extern os_typespec * os_alloc_typespec(char * type_string);

Allocates and constructs an **os_typespec** in transient memory.

The **os_typespec** represents the type named by the **type_string.** The **type_string** can name a built-in type (like **char** or **int**), a struct (leave off the reserved word "**struct**", as in "**part**"), a pointer-to-built-in (like **char***), or a pointer-to-class (like **part***). This means, for example, that "**part****" is not allowed — use "**void***" instead. No spaces are allowed in the **type_string**, so, for example, the name should contain no spaces before a “*****” character.

Examples

```
os_typespec *part_type = os_alloc_typespec("line#");
os_typespec *part_ptr_type = os_alloc_typespec("part*");
```

You must mark each struct that your application uses in a persistent context by performing **OS_MARK_SCHEMA_TYPE()** on it. Call this macro once for each type of struct your application reads from or writes to persistent memory. The calls should appear inside a dummy function definition. The struct name is the argument, and should appear without quotation marks and without the reserved word "**struct**". See *ObjectStore Building C++ Interface Applications*.

```
extern void* os_database_alloc(
        os_typespec*, /* what is it */
        os_int32, /* how many? */
        os_database *, /* which db */
        os_boolean make_array /* for count==1 */
```

Allocates persistent storage of the specified type in the specified database. To allocate an array, specify the number of elements in the array with the **os_int32** argument, and pass 1 to the **os_ boolean** argument. To allocate a nonarray, pass 1 to the **os_int32** argument and pass 0 to the **os_boolean** argument. This function cannot be used for transient allocation.

```
extern void* os_segment_alloc(
        os_typespec *, /* what is it */
        os_int32, /* how many? */
        os_segment * /* which seg */
        os_boolean make_array /* for count==1 */);
```

Allocates persistent storage of the specified type in the specified segment. To allocate an array, specify the number of elements in the array with the **os_int32** argument, and pass 1 to the **os_ boolean** argument. To allocate a nonarray, pass 1 to the **os_int32** argument and pass 0 to the **os_boolean** argument. This function cannot be used for transient allocation.

```
extern void* os_object_cluster_alloc(
        os_typespec *,/* what is it */
        os_int32,/* how many? * /
        os_object_cluster */* which cluster */
        os_boolean make_array /* for count==1 */);
```

Allocates persistent storage of the specified type in the specified object cluster. To allocate an array, specify the number of elements in the array with the **os_int32** argument, and pass 1 to the **os_ boolean** argument. To allocate a nonarray, pass 1 to the **os_int32** argument and pass 0 to the **os_boolean** argument. This function cannot be used for transient allocation.

Note that these allocation functions should not be used for allocating instances of ObjectStore types (such as **os_collection**). ObjectStore types have class-specific allocation functions that should be used instead.

```
extern void objectstore_delete(void *);
```

Deletes the specified value from storage. The argument must point to the beginning of an object's storage. If it points to the middle of an object, the exception **err_invalid_deletion** is signaled. This might happen, for example, if the argument points to the fifth element of a persistent array.

Reallocating Storage

objectstore_realloc makes it easier to convert C programs to ObjectStore. The C++ library interface version is **objectstore::change_array_length()**.

extern char * objectstore_realloc(
);

Takes one **char *** pointer argument and a size argument, and returns a **char *** pointer value. If the pointer argument is a pointer to transient memory, ordinary UNIX **realloc** is called and the result is returned.

If the argument is a pointer to persistent memory, it must point to the beginning of an array of persistent objects. Furthermore, the size argument must be an exact positive multiple of the size of the type of the objects, in bytes. **objectstore_realloc** does the semantic equivalent of allocating a new array of objects, copying the old objects to the new objects (bit-wise, that is; no C++ constructors are run), initializing the rest of the new array to zero, and deallocating the original array. It returns a pointer to the new array. The new pointer might or might not equal the old pointer.

The caller should not retain and later dereference any copies of the original pointer (just as if **objectstore_delete** had been called). The *compatibility feature* of **realloc** that allows it to be called on the most recently freed block is not supported for persistent objects.

Transaction Macros

All access to persistent data must take place within a transaction. (Some manipulation of databases themselves, such as opening and closing them, need not take place within a transaction.)

The macros **OS_BEGIN_TXN** and **OS_END_TXN** start and end transactions. Note that

- As shown below, **OS_BEGIN_TXN** and **OS_END_TXN** both require a *unique tag*, a C identifier that is similar to a tag on a statement. The *unique_tag* argument must be the same for the macros that begin and end each transaction.

- In place of the C++ enumerators **os_transaction::update** and **os_transaction::read_only,** the C library interface uses the C enumerators **os_transaction_update** and **os_transaction_read_only.**

Calls to the macros have the following form:

OS_BEGIN_TXN(*unique_tag***,***excp_ptr***,***txn_type***)**
OS_END_TXN(*unique_tag***)**

unique_tag is a C identifier. The same tag must be used at both ends of a transaction, and no other transaction in the program can use the same tag.

excp_ptr points either to **0** (meaning the transaction committed), or to a **tix_exception***, which indicates the nature of the abort.

txn_type is either **os_transaction_update** (specifying that the transaction is to perform updates to persistent storage), or **os_transaction_read_only** (specifying no update to persistent storage) or **os_transaction_abort_only**.

objectstore Functions

The functions in this section correspond to members of the system-supplied ObjectStore class **objectstore** and to ObjectStore system utilities documented in *Managing ObjectStore*.

objectstore_acquire_lock

```
extern os_lock_timeout_exception *objectstore_acquire_lock(
        void *addr,
        os_lock_type access,
        os_int32 milliseconds,
        os_unsigned_int32 size);
```

```
extern os_lock_timeout_exception *acquire_lock(
      os_database *db,
      os_lock_type access,
      os_int32 milliseconds);
```

```
extern os_lock_timeout_exception *acquire_lock(
      os_segment *seg,
      os_lock_type access,
      os_int32 milliseconds);
```

See **objectstore::acquire_lock()** on page 11.

objectstore_delete

```
extern void objectstore_delete(void *);
```

See **void ::operator delete()** on page 366.

objectstore_disc_swap_bytes

```
extern void objectstore_disc_swap_bytes(
        char *,
        char *,
        os_int32);
```

See **objectstore::discriminant_swap_bytes()** on page 17.

objectstore_full_initialize

```
extern void objectstore_full_initialize();
```

See **objectstore::initialize()** on page 26.

objectstore_get_all_servers

```
extern void objectstore_get_all_servers(
        os_int32,
        os_server**,
        os_int32*);
```

See **objectstore::get_all_servers()** on page 20.

objectstore_get_application_schema_pathname

```
extern const char * objectstore_get_application_schema_
pathname();
```

See **objectstore::get_application_schema_pathname()** on page 21.

objectstore_get_application_schema_pathname

```
extern char *objectstore_get_application_schema_pathname( );
```

See **objectstore::get_application_schema_pathname()** on page 21.

objectstore_get_auto_open_mode

```
void objectstore_get_auto_open_mode(
    auto_open_mode_enum *mode,
    os_fetch_policy *fp,
    os_int32 *bytes);
```

See **objectstore::get_auto_open_mode()** on page 21.

objectstore_get_auto_open_read_whole_segment_mode

```
extern os_boolean
    objectstore_get_auto_open_read_whole_segment_mode();
```

See **objectstore::get_auto_open_mode()** on page 21.

objectstore_get_cache_size

```
extern os_unsigned_int32 objectstore_get_cache_size();
```

See **objectstore::get_cache_size()** on page 22.

objectstore_get_incremental_schema_installation

```
extern os_boolean
    objectstore_get_incremental_schema_installation();
```

See **objectstore::get_incremental_schema_installation()** on page 22.

objectstore_get_lock_status

```
extern os_lock_type
    objectstore_get_lock_status(void *address);
```

See **objectstore::get_lock_status()** on page 23.

objectstore_get_n_servers

```
extern os_int32 objectstore_get_n_servers();
```

See **objectstore::get_n_servers()** on page 23.

objectstore_get_null_illegal_pointers

```
extern os_boolean objectstore_get_null_illegal_pointers();
```

See **objectstore::get_null_illegal_pointers()** on page 23.

objectstore_get_object_range

```
extern void objectstore_get_object_range(
        void const *,/* address */
        void **,/* base_address */
        os_unsigned_int32 */* size */);
```

See **objectstore::get_object_range()** on page 23.

objectstore_get_object_total_range

```
extern void objectstore_get_object_total_range(
        void const *,/* address */
        void **,/* base_address */
        os_unsigned_int32 *,/* size */
        void **,/* header_address */
        os_unsigned_int32 */* total_size */);
```

See **objectstore::get_object_range()** on page 23.

objectstore_get_opt_cache_lock_mode

```
extern os_boolean objectstore_get_opt_cache_lock_mode();
```

See **objectstore::get_opt_cache_lock_mode()** on page 24.

objectstore_get_page_size

```
extern os_unsigned_int32 objectstore_get_page_size();
```

See **objectstore::get_page_size()** on page 24.

objectstore_get_pointer_numbers

```
extern void objectstore_get_pointer_numbers(
        void *,/* address *
        os_unsigned_int32*,/* number 1 */
        os_unsigned_int32*,/* number 2 */
        os_unsigned_int32*/* number 3 */);
```

See **objectstore::get_pointer_numbers()** on page 24.

objectstore_get_readlock_timeout

```
extern os_int32 objectstore_get_readlock_timeout();
```

See **objectstore::get_readlock_timeout()** on page 24.

objectstore_get_retain_persistent_addresses

```
extern os_boolean objectstore_get_retain_persistent_addresses();
```

See **objectstore::get_retain_persistent_addresses()** on page 25.

objectstore_get_simple_auth_ui

```
extern void objectstore_get_simple_auth_ui(
            os_simple_auth_ui_t*,
            void**);
```

See **objectstore::get_simple_auth_ui()** on page 25.

objectstore_delete_function

```
extern objectstore_delete_function
    objectstore_get_transient_delete_function();
```

See **objectstore::get_transient_delete_function()** on page 25.

objectstore_get_writelock_timeout

```
extern os_int32 objectstore_get_writelock_timeout();
```

See **objectstore::get_writelock_timeout()** on page 25.

objectstore_hidden_write

```
extern void objectstore_hidden_write(
    char *,
    char *,
    os_unsigned_int32);
```

See **objectstore::hidden_write()** on page 26.

objectstore_is_lock_contention

```
extern os_boolean objectstore_is_lock_contention();
```

See **objectstore::is_lock_contention()** on page 27.

objectstore_is_persistent

```
extern os_boolean objectstore_is_persistent(
    void */* addr */);
```

See **objectstore::is_persistent()** on page 28.

objectstore_realloc

```
extern char*objectstore_realloc(
    char *,
    os_unsigned_int32);
```

See **objectstore::change_array_length()** on page 14.

objectstore_retain_persistent_addresses

```
extern void objectstore_retain_persistent_addresses();
```

See **objectstore::retain_persistent_addresses()** on page 31.

objectstore_return_all_pages

```
extern os_unsigned_int32 objectstore_return_all_pages();
```

See **objectstore::return_all_pages()** on page 31.

objectstore_return_memory

```
extern os_unsigned_int32 objectstore_return_memory(
    void *address,
    os_unsigned_int32 length,
    os_boolean evict_now);
```

See **objectstore::return_memory()** on page 31.

objectstore_set_aid

```
extern void objectstore_set_aid(
    os_int32 aid);
```

See **objectstore::set_aid()**.

objectstore_set_always_ignore_illegal_pointers

```
void objectstore_set_always_ignore_illegal_pointers(
            os_boolean new_val);
```

See **objectstore::set_always_ignore_illegal_pointers()** on page 32.

objectstore_set_always_null_illegal_pointers

```
void objectstore_set_always_null_illegal_pointers(
    os_boolean new_val);
```

See **objectstore::set_always_null_illegal_pointers()** on page 32.

objectstore_set_application_schema_pathname

```
extern void objectstore_set_application_schema_pathname(
    const char * path);
```

See **objectstore::set_application_schema_pathname()** on page 32.

objectstore_set_auto_open_mode

```
void objectstore_set_auto_open_mode(
    auto_open_mode_enum mode,
    os_fetch_policy fp,
    os_int32 bytes);
```

See **objectstore::set_auto_open_mode()** on page 33.

objectstore_set_auto_open_read_whole_segment_mode

```
extern void
    objectstore_set_auto_open_read_whole_segment_mode(
    os_boolean val);
```

See **objectstore::set_auto_open_mode()** on page 33.

objectstore_set_cache_size

```
extern void objectstore_set_cache_size(
        os_unsigned_int32 size);
```

See **objectstore::set_cache_size()** on page 35.

objectstore_set_client_name

```
extern void objectstore_set_client_name(
        char */* name */);
```

See **objectstore::set_client_name()** on page 35.

objectstore_set_commseg_size

```
extern void objectstore_set_commseg_size(
        os_unsigned_int32 size);
```

See **objectstore::set_commseg_size()** on page 35.

objectstore_set_current_schema_key

```
extern void objectstore_set_current_schema_key(
        os_unsigned_int32 key_low,
        os_unsigned_int32 key_high);
```

See **objectstore::set_current_schema_key()** on page 36.

objectstore_set_eviction_batch_size

```
extern void objectstore_set_eviction_batch_size(
        os_unsigned_int32/* size, in bytes */);
```

See **objectstore::set_eviction_batch_size()** on page 36.

objectstore_set_eviction_pool_size

```
extern void objectstore_set_eviction_pool_size(
        os_unsigned_int32/* size, in bytes */);
```

See **objectstore::set_eviction_pool_size()** on page 37.

objectstore_set_handle_transient_faults

```
extern void objectstore_set_handle_transient_faults(
         os_boolean);
```

See **objectstore::set_handle_transient_faults()** on page 37.

objectstore_set_incremental_schema_installation

```
extern void objectstore_set_incremental_schema_installation(
        os_boolean);
```

See **objectstore::set_incremental_schema_installation()** on page 37.

objectstore_set_mapped_communications

```
extern void objectstore_set_mapped_communications(
        os_boolean);
```

See **objectstore::set_mapped_communications()** on page 38.

objectstore_set_null_illegal_pointers

```
extern void objectstore_set_null_illegal_pointers(
os_boolean new_val);
```

See **objectstore::set_null_illegal_pointers()** on page 38.

objectstore_set_opt_cache_lock_mode

```
extern void objectstore_set_opt_cache_lock_mode(
        os_boolean);
```

See **objectstore::set_opt_cache_lock_mode()** on page 39.

objectstore_set_readlock_timeout

```
extern void objectstore_set_readlock_timeout(
        os_int32 milliseconds);
```

See **objectstore::set_readlock_timeout()** on page 39.

objectstore_set_reserve_as_mode

```
extern void objectstore_set_reserve_as_mode(
          os_boolean/* new mode */);
```

See **objectstore::set_reserve_as_mode()** on page 39.

objectstore_set_simple_auth_ui

```
extern void objectstore_set_simple_auth_ui(
          os_simple_auth_ui_t,
          void*);
```

See **objectstore::set_simple_auth_ui()** on page 39.

objectstore_set_transient_delete_function

```
extern void objectstore_set_transient_delete_function(
          objectstore_delete_function f);
```

See **objectstore::set_transient_delete_function()** on page 41.

objectstore_set_writelock_timeout

```
extern void objectstore_set_writelock_timeout(
          os_int32 milliseconds);
```

See **objectstore::set_writelock_timeout()** on page 42.

os_database Functions

The functions in this section correspond to members of the system-supplied ObjectStore class **os_database**.

Persistent data can be accessed only if the database in which it resides is open. Databases are created, destroyed, opened, and closed with database functions. These functions can be called either within or outside a transaction.

Each database retrieved by a given process has an associated *open count* for that process. The function **database_open()** increments the open count by 1, and the function **database_close()** decrements the open count by 1. When the open count for a process increases from 0 to 1, the database becomes open for that process. A database becomes closed for a process when its open count becomes 0.

When a process terminates, any databases left open are automatically closed.

In the functions below, the argument **database *** corresponds to the implicit **this** pointer found in members of class **os_database**.

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_database_alloc

```
extern void *
os_database_alloc(
        os_typecode *,/* what is it */
        os_int32,/* how many? */
        os_database *,/* which db */
        os_boolean make_array /* for count==1 */);
```

See page 399.

os_database_allow_external_pointers

```
extern void
os_database_allow_external_pointers(
        os_database *,
        os_boolean);
```

See **os_database::allow_external_pointers()** on page 72.

os_database_change_database_reference

```
extern void os_database_change_database_reference(
        os_database *,
        os_database_reference *,
        os_database_reference *);
```

See **os_database::change_database_reference()** on page 73.

os_database_change_schema_key

```
extern void os_database_change_sc9hema_key(
   os_database *db,
        os_unsigned_int32 old_key_low,
        os_unsigned_int32 old_key_high,
        os_unsigned_int32 new_key_low,
        os_unsigned_int32 new_key_high);
```

See **os_database::change_schema_key()** on page 73.

os_database_close

```
extern void os_database_close(
        os_database *);
```

See **os_database::close()** on page 74.

os_database_create

```
extern os_database *os_database_create(
        char *,/* pathname */
        os_int32,/* mode */
        os_boolean/* if_exists_overwrite */);
```

See **os_database::create()** on page 74.

os_database_create_root

```
extern os_database_root *os_database_create_root(
        os_database * db,
        char *name);
```

See **os_database::create_root()** on page 76.

os_database_create_segment

```
extern os_segment *os_database_create_segment(
        os_database *);
```

See **os_database::create_segment()** on page 76.

os_database_destroy

```
extern void os_database_destroy(
        os_database *);
```

See **os_database::destroy()** on page 77.

os_database_find_root

```
extern os_database_root *os_database_find_root(
         os_database *db,
         char *name);
```

See **os_database::find_root()** on page 78.

os_database_freeze_schema_key

```
extern void os_database_freeze_schema_key(
    os_database *db,
        os_unsigned_int32 key_low,
        os_unsigned_int32 key_high);
```

See **os_database::freeze_schema_key()** on page 78.

os_database_get_all_databases

```
extern void os_database_get_all_databases(
        os_int32,/* max db's to return */
        os_database **,/* where to return them */
        os_int32 */* how many returned */);
```

See **os_database::get_all_databases()** on page 79.

os_database_get_all_roots

```
extern void os_database_get_all_roots(
        os_database *,
        os_int32,/* max to return */
        os_database_root **,/* where to return them */
        os_int32 */* n returned */);
```

See **os_database::get_all_roots()** on page 80.

os_database_get_all_segments

```
extern void os_database_get_all_segments(
        os_database *,
        os_int32,/* max segs to return */
        os_segment **,/* where to return them */
        os_int32 */* how many returned */);
```

See **os_database::get_all_segments()** on page 80.

os_database_get_all_segments_and_permissions

```
extern void os_database_get_all_segments_and_permissions(
        os_database *,
        os_int32,/* max segs to return */
        os_segment **,/* where to return them */
        os_segment_access **,/* access controls */
        os_int32 */* how many returned */);
```

See **os_database::get_all_segments_and_permissions()** on page 80.

os_database_get_application_info

```
extern void *os_database_get_application_info(
        os_database *);
```

See **os_database::get_application_info()** on page 81.

os_database_get_database_references

```
extern void os_database_get_database_references(
        os_database *,
        os_int32 *,
        os_database_reference ***);
```

See **os_database::get_database_references()** on page 81.

os_database_get_default_check_illegal_pointers

```
extern os_boolean os_database_get_default_check_illegal_
pointers(
        os_database *);
```

See **os_database::get_default_check_illegal_pointers()** on page 81.

os_database_get_default_lock_whole_segment

```
extern objectstore_lock_optionos_database_get_default_lock_
whole_segment(
        os_database *);
```

See **os_database::get_default_lock_whole_segment()** on page 81.

os_database_get_default_null_illegal_pointers

```
extern os_boolean os_database_get_default_null_illegal_pointers(
        os_database *);
```

See **os_database::get_default_null_illegal_pointers()** on page 82.

os_database_get_default_read_whole_segment

```
extern os_boolean os_database_get_default_read_whole_segment(
        os_database *);
```

See **os_database::get_default_read_whole_segment()**.

os_database_get_default_segment

```
extern os_segment *os_database_get_default_segment(
        os_database *);
```

See **os_database::get_default_segment()** on page 82.

os_database_get_default_segment_size

```
extern os_unsigned_int32 os_database_get_default_segment_size(
        os_database *);
```

See **os_database::get_default_segment_size()** on page 82.

os_database_get_fetch_policy

```
extern void os_database_get_fetch_policy(
        os_database *,
        os_fetch_policy *,
        os_int32 *);
```

See **os_database::get_fetch_policy()** on page 83.

os_database_get_host_name

```
extern char* os_database_get_host_name(
        os_database *);
```

See **os_database::get_host_name()** on page 83.

os_database_get_id

```
extern os_database_id* os_database_get_id(
        os_database *);
```

See **os_database::get_id()** on page 83.

os_database_get_incremental_schema_installation

```
extern os_boolean os_database_get_incremental_schema_
installation(
        os_database *);
```

See **os_database::get_incremental_schema_installation()** on page 83.

os_database_get_lock_whole_segment

```
extern objectstore_lock_option os_database_get_lock_whole_
segment(
        os_database *);
```

See **os_database::get_lock_whole_segment()** on page 83.

os_database_get_n_databases

```
extern os_int32 os_database_get_n_databases();
```

See **os_database::get_n_databases()** on page 84.

os_database_get_n_roots

```
extern os_int32 os_database_get_n_roots(
        os_database */* db */);
```

See **os_database::get_n_roots()** on page 84.

os_database_get_n_segments

```
extern os_int32 os_database_get_n_segments(
        os_database */* db */);
```

See **os_database::get_n_segments()** on page 84.

os_database_get_null_illegal_pointers

```
extern os_boolean os_database_get_null_illegal_pointers(
    os_database *);
```

os_database_get_opt_cache_lock_mode

```
extern os_boolean os_database_get_opt_cache_lock_mode(
    os_database *);
```

See **os_database::get_opt_cache_lock_mode()** on page 84.

os_database_get_pathname

```
extern char*os_database_get_pathname(
        os_database *);
```

See **os_database::get_pathname()** on page 84.

os_database_get_readlock_timeout

```
extern os_int32 os_database_get_readlock_timeout(
        os_database *db);
```

See **os_database::get_readlock_timeout()** on page 85.

os_database_get_relative_directory

```
extern char *os_database_get_relative_directory(
          os_database *);
```

See **os_database::get_relative_directory()** on page 85.

os_database_get_sector_size

```
extern os_int32 os_database_get_sector_size(
          os_database *);
```

See **os_database::get_sector_size()** on page 86.

os_database_get_segment

```
extern os_segment *os_database_get_segment(
          os_database *,
          os_unsigned_int32 segment_number);
```

See **os_database::get_segment()** on page 86.

os_database_get_transient_database

```
extern os_database *os_database_get_transient_database();
```

See **os_database::get_transient_database()** on page 86.

os_database_get_writelock_timeout

```
extern os_int32 os_database_get_writelock_timeout(
   os_database *db);
```

See **os_database::get_writelock_timeout()** on page 86.

os_database_is_open

```
extern os_boolean os_database_is_open(
   os_database *);
```

See **os_database::is_open()** on page 87.

os_database_is_open_mvcc

```
extern os_boolean os_database_is_open_mvcc(
   os_database *);
```

See **os_database::open_mvcc()** on page 89.

os_database_is_open_read_only

```
extern os_boolean os_database_is_open_read_only(
   os_database *);
```

See **os_database::is_open_read_only()** on page 87.

os_database_is_writable

```
extern os_boolean os_database_is_writable(
    os_database */* db */);
```

See **os_database::is_writable()** on page 87.

os_database_lookup

```
extern os_database *os_database_lookup(
    char *,/* pathname */
    os_int32/* mode */);
```

See **os_database::lookup()** on page 88.

os_database_lookup_open

```
extern os_database *os_database_lookup_open(
    char *,/* pathname */
    os_boolean,/* read_only */
    os_boolean/* create_mode */);
```

See **os_database::open()** on page 88.

os_database_lookup_open_mvcc

```
extern os_database *os_database_lookup_open_mvcc(
    char */* pathname */);
```

See **os_database::open_mvcc()** on page 89.

os_database_of

```
extern os_database *os_database_of(
    void */* location */);
```

See **os_database::of()** on page 88.

os_database_open

```
extern void os_database_open(
        os_database *,
        os_boolean/* read_only */);
```

See **os_database::open()** on page 88.

os_database_open_mvcc

```
extern void os_database_open_mvcc(
        os_database *);
```

See **os_database::open_mvcc()** on page 89.

os_database_set_access_hooks

```
extern void os_database_set_access_hooks(
        os_database *,
        const char *,
        os_access_hook,
        os_access_hook,
        void *,
        os_access_hook *,
        os_access_hook *,
        void **);
```

See **os_database::set_access_hooks()** on page 91.

os_database_set_application_info

```
extern void os_database_set_application_info(
        os_database *,
        void *);
```

See **os_database::set_application_info()** on page 93.

os_database_set_check_illegal_pointers

```
extern void os_database_set_check_illegal_pointers(
        os_database *,
        os_boolean);
```

See **os_database::set_check_illegal_pointers()** on page 93.

os_database_set_default_check_illegal_pointers

```
extern void os_database_set_default_check_illegal_pointers(
        os_database *,
        os_boolean);
```

See **os_database::set_default_check_illegal_pointers()** on page 94.

os_database_set_default_lock_whole_segment

```
extern void os_database_set_default_lock_whole_segment(
        os_database *,
        objectstore_lock_option);
```

See **os_database::set_default_lock_whole_segment()** on page 94.

os_database_set_default_null_illegal_pointers

```
extern void os_database_set_default_null_illegal_pointers(
        os_database *,
        os_boolean);
```

See **os_database::set_default_null_illegal_pointers()** on page 94.

os_database_set_default_segment

```
extern void os_database_set_default_segment(
        os_database *,
        os_segment *);
```

See **os_database::set_default_segment()** on page 94.

os_database_set_default_segment_size

```
extern void os_database_set_default_segment_size(
        os_database *,
        os_unsigned_int32);
```

See **os_database::set_default_segment_size()** on page 95.

os_database_set_fetch_policy

```
extern void os_database_set_fetch_policy(
        os_database *,
        os_fetch_policy,
        os_int32);
```

See **os_database::set_fetch_policy()** on page 95.

os_database_set_incremental_schema_installation

```
extern void os_database_set_incremental_schema_installation(
        os_database *,
        os_boolean);
```

See **os_database::set_incremental_schema_installation()** on page 96.

os_database_set_lock_whole_segment

```
extern void os_database_set_lock_whole_segment(
        os_database *,
        objectstore_lock_option);
```

See **os_database::set_lock_whole_segment()** on page 97.

os_database_set_new_id

```
extern os_boolean os_database_set_new_id(
        os_database *);
```

See **os_database::set_new_id()** on page 97.

os_database_set_null_illegal_pointers

```
extern void os_database_set_null_illegal_pointers(
        os_database *,
        os_boolean);
```

See **os_database::set_null_illegal_pointers()** on page 97.

os_database_set_opt_cache_lock_mode

```
extern void os_database_set_opt_cache_lock_mode(
        os_database *,
        os_boolean);
```

See **os_database::set_opt_cache_lock_mode()** on page 98.

os_database_set_readlock_timeout

```
extern void os_database_set_readlock_timeout(
        os_database *db,
        os_int32 milliseconds);
```

See **os_database::set_readlock_timeout()** on page 98.

os_database_set_relative_directory

```
extern void os_database_set_relative_directory(
        os_database *,
        char *);
```

See **os_database::set_relative_directory()** on page 98.

os_database_set_writelock_timeout

```
extern void os_database_set_writelock_timeout(
        os_database *db,
        os_int32 milliseconds);
```

See **os_database::set_writelock_timeout()** on page 99.

os_database_size

```
extern os_unsigned_int32 os_database_size(
        os_database *);
```

See **os_database::size()** on page 99.

os_database_size_in_sectors

```
extern os_unsigned_int32 os_database_size_in_sectors(
        os_database *);
```

See **os_database::size_in_sectors()** on page 100.

os_database_time_created

```
extern os_unixtime_t os_database_time_created(
        os_database *);
```

See **os_database::time_created()** on page 100.

os_database_reference Functions

os_database_reference_delete_array

```
extern void os_database_reference_delete_array(
        os_int32,
        os_database_reference **);
```

Deletes the specified database reference array.

os_database_reference_equal

```
extern os_boolean os_database_reference_equal(
        os_database_reference *,
        os_database_reference *);
```

See **os_database_reference::operator ==()** on page 101.

os_database_reference_free

```
extern void os_database_reference_free(
        os_database_reference *);
```

Deletes the specified database reference.

os_database_reference_get_name

```
extern char * os_database_reference_get_name(
        os_database_reference *);
```

See **os_database_reference::get_name()** on page 101.

os_new_database_reference_name

```
extern database_reference * os_new_database_reference_name(
   char *);
```

Allocates a new database reference.

os_database_root Functions

Each database records the association between its entry points and their persistent names with the **database_root** functions.

os_database_root_find

```
extern database_root * os_database_root_find(
    char *,/* name */
    database *);
```

See **os_database_root::find()** on page 103.

os_database_root_free

```
extern void os_database_root_free(
    database_root *);
```

See **os_database_root::~os_database_root()** on page 103.

os_database_root_get_name

```
extern char * os_database_root_get_name(
    database_root *);
```

See **os_database_root::get_name()** on page 103.

os_database_root_get_typespec

```
extern os_typespec * os_database_root_get_typespec(
    database_root *);
```

See **os_database_root::get_typespec()** on page 103.

os_database_root_get_value

```
extern void * os_database_root_get_value(
    database_root *,
    os_typespec *);
```

See **os_database_root::get_value()** on page 103.

os_database_root_set_value

```
extern void os_database_root_set_value(
    database_root *,
    void *,/* new_value */
    os_typespec */* new_typespec_arg */);
```

See **os_database_root::set_value()** on page 104.

os_dbutil Functions

The C library interface contains functions analogous to those of the class **os_dbutil** in the ObjectStore Class Library.

os_dbutil_chgrp

```
extern void os_dbutil_chgrp(
        const char* /* pathname */,
        const char* /* gname */);
```

See **os_dbutil::chgrp()** on page 106.

os_dbutil_chmod

```
extern void os_dbutil_chmod(
        const char* /* pathname */,
        os_unsigned_int32 /* mode */);
```

See **os_dbutil::chmod()** on page 106.

os_dbutil_chown

```
extern void os_dbutil_chown(
        const char* /* pathname */,
        const char* /* uname */
        );
```

See **os_dbutil::chown()** on page 106.

os_dbutil_close_all_connections

```
extern void os_dbutil_close_all_connections();
```

See **os_dbutil::close_all_server_connections()** on page 107.

os_dbutil_close_all_server_connections

```
extern void os_dbutil_close_all_server_connections();
```

See **os_dbutil::close_all_server_connections()** on page 107.

os_dbutil_close_server_connection

```
extern void os_dbutil_close_server_connection(
        const char* /* hostname */
        );
```

See **os_dbutil::close_server_connection()** on page 107.

os_dbutil_cmgr_log

```
extern char *os_dbutil_cmgr_log(
        const char * /* hostname */,
        os_int32 /* cm_version_number */
        );
```

os_dbutil_cmgr_remove_file

```
extern char *os_dbutil_cmgr_remove_file(
        const char* /* hostname */,
        os_int32 /* cm_version_number */
        );
```

See **os_dbutil::cmgr_remove_file()** on page 107.

os_dbutil_cmgr_shutdown

```
extern char *os_dbutil_cmgr_shutdown(
        const char* /* hostname */,
        os_int32 /* cm_version_number */
        );
```

See **os_dbutil::cmgr_shutdown()** on page 107.

os_dbutil_cmgr_stat

```
extern void os_dbutil_cmgr_stat(
        const char* /* hostname */,
        os_int32 /* cm_version_number */,
        os_cmgr_stat* /* cmstat_data */
        );
```

See **os_dbutil::cmgr_stat()** on page 108.

os_dbutil_compare_schemas

```
extern os_boolean os_dbutil_compare_schemas(
        const os_database* /* db1 */,
        const os_database* /* db2 */,
        os_boolean /* verbose */
        );
```

See **os_dbutil::compare_schemas()** on page 110.

os_dbutil_copy_database

```
extern os_boolean os_dbutil_copy_database(
        const char* /* src_database_name */,
        const char* /* dest_database_name */
        );
```

See **os_dbutil::copy_database()** on page 110.

os_dbutil_delete_os_rawfs_entries

```
extern void os_dbutil_delete_os_rawfs_entries(
        os_rawfs_entry *stat_entry,
        os_boolean is_array
);
```

os_dbutil_disk_free

```
extern void os_dbutil_disk_free(
        const char* /* hostname */,
        os_free_blocks * /* free_blocks */
        );
```

See **os_dbutil::disk_free()** on page 110.

os_dbutil_expand_global

```
extern char **os_dbutil_expand_global(
        const char *, /* glob */
        os_unsigned_int32 * /* n_entries */
        );
```

See **os_dbutil::expand_global()** on page 111.

os_dbutil_get_client_name

```
extern char *os_dbutil_get_client_name();
```

See **os_dbutil::get_client_name()** on page 111.

os_dbutil_get_sector_size

```
extern os_unsigned_int32 os_dbutil_get_sector_size();
```

See **os_dbutil::get_sector_size()** on page 112.

os_dbutil_initialize

```
extern void os_dbutil_initialize();
```

See **os_dbutil::initialize()** on page 112.

os_dbutil_list_directory

```
os_rawfs_entry **os_dbutil_list_directory(
        const char* /* path */,
        os_unsigned_int32 * /* n_entries */
        );
```

See **os_dbutil::list_directory()** on page 112.

os_dbutil_make_link

```
extern void os_dbutil_make_link(
        const char* /* target_name */,
        const char* /* link_name */
        );
```

See **os_dbutil::make_link()** on page 112.

os_dbutil_mkdir

```
extern void os_dbutil_mkdir(
        const char* /* path */,
        const os_unsigned_int32 /* mode */,
        os_boolean /* create_missing_dirs */
        );
```

See **os_dbutil::mkdir()** on page 113.

os_dbutil_ossize

```
extern os_int32 os_dbutil_ossize(
        const char* /* pathname */,
        const os_size_options * /* options */
        );
```

See **os_dbutil::ossize()** on page 113.

os_dbutil_osverifydb

```
extern os_unsigned_int32 os_dbutil_osverifydb(
        const char* /* pathname */,
        os_verifydb_options* /* options */
        );
```

See **os_dbutil::osverifydb()** on page 114.

os_dbutil_osverifydb_one_segment

```
extern os_unsigned_int32 os_dbutil_osverifydb_one_segment(
        const char* /* pathname */,
        os_unsigned_int32 /* seg num */,
        os_unsigned_int32 /* starting offset*/,
        os_unsigned_int32 /* ending offset*/,
        os_verifydb_options* /* options */
        );
```

See **os_dbutil::osverifydb_one_segment()** on page 116.

os_dbutil_rehost_all_links

```
extern void os_dbutil_rehost_all_links(
      const char* /* server_host */,
          const char* /* old_host */,
          const char* /* new_host */
          );
```

See **os_dbutil::rehost_all_links()** on page 117.

os_dbutil_rehost_link

```
extern void os_dbutil_rehost_link(
          const char* /* pathname */,
          const char* /* new_host */
         );
```

See **os_dbutil::rehost_link()** on page 117.

os_dbutil_remove

```
extern void os_dbutil_remove(
             const char* /* path */
             );
```

See **os_dbutil::remove()** on page 118.

os_dbutil_rename

```
extern void os_dbutil_rename(
             const char* /* source */,
             const char* /* target */
             );
```

See **os_dbutil::rename()** on page 118.

os_dbutil_rmdir

```
extern void os_dbutil_rmdir(
             const char* /* path */
             );
```

See **os_dbutil::rmdir()** on page 118.

os_dbutil_set_application_schema_path

```
extern char *os_dbutil_set_application_schema_path(
             const char* /* exe_path */,
             const char* /* db_path */
             );
```

See **os_dbutil::set_application_schema_path()** on page 119.

os_dbutil_set_client_name

```
extern void os_dbutil_set_client_name(
             const char * /* name */
```

```
        );
```

See **os_dbutil::set_client_name()** on page 119.

os_dbutil_stat

```
os_rawfs_entry *os_dbutil_stat(
        const char* /* path */,
        const os_boolean /* b_chase_links */
        );
```

See **os_dbutil::stat()** on page 119.

os_dbutil_svr_checkpoint

```
extern os_boolean os_dbutil_svr_checkpoint(
        const char* /* server_host */
        );
```

See **os_dbutil::svr_checkpoint()** on page 119.

os_dbutil_svr_client_kill

```
extern os_boolean os_dbutil_svr_client_kill(
        const char* /* server_host */,
        os_int32 /* client_pid */,
        const char* /* client_name */,
        const char* /* client_hostname */,
        os_boolean /* all */,
        os_int32* /* status */
        );
```

See **os_dbutil::svr_client_kill()** on page 120.

os_dbutil_svr_ping

```
extern char *os_dbutil_svr_ping(
        const char* /* server_host */,
        os_svr_ping_state* /* state */
        );
```

See **os_dbutil::svr_ping()** on page 120.

os_dbutil_svr_shutdown

```
extern os_boolean os_dbutil_svr_shutdown(
        const char* /* server_host */
        );
```

See **os_dbutil::svr_shutdown()** on page 121.

os_dbutil_svr_stat

```
extern void os_dbutil_svr_stat(
        const char* /* server_host */,
        os_int32 /* requests */,
        os_svr_stat * /* svr_stat_data */
        );
```

See **os_dbutil::svr_stat()** on page 121.

os_size_options_allocate

```
os_size_options* os_size_options_allocate
```

Allocates an appropriately initialized **os_size_options** struct.

See **os_dbutil::ossize()** on page 113.

os_size_options_delete

```
void os_size_options_delete(
        os_size_options* /* options */,
        os_boolean /* is_array */
);
```

Deletes an **os_size_options** struct.

os_svr_stat_client_info_delete

```
void os_svr_stat_client_info_delete(
        os_svr_stat_client_info* /* client_info */,
        os_boolean /* is_array */
);
```

Deletes an **os_svr_stat_client_info** struct.

os_svr_stat_client_meters_delete

```
void os_svr_stat_client_meters_delete(
        os_svr_stat_client_meters* /* client_meter */,
        os_boolean /* is_array */
);
```

Deletes an **os_svr_stat_client_meters** struct.

os_svr_stat_client_process_delete

```
void os_svr_stat_client_process_delete(
        os_svr_stat_client_process* /* client_process */,
        os_boolean /* is_array */
);
```

Deletes an **os_svr_stat_client_process** struct.

os_svr_stat_client_state_delete

```
void os_svr_stat_client_state_delete(
        os_svr_stat_client_state* /* client_state */,
        os_boolean /* is_array */
);
```

Deletes an **os_svr_stat_client_state** struct.

os_svr_stat_delete

```
    os_svr_stat* svr_stat
    os_boolean is_array
);
```

Deletes an **os_svr_stat_svr_header** struct.

os_svr_stat_svr_header_delete

```
void os_svr_stat_svr_header_delete(
        os_svr_stat_svr_header* /* svr_header */,
        os_boolean /* is_array */
);
```

Deletes an **os_svr_stat_svr_header** struct.

os_svr_stat_svr_meters_delete

```
void os_svr_stat_svr_meters_delete(
        os_svr_stat_svr_meters* /* svr_meter */,
        os_boolean /* is_array */
);
```

Deletes an **os_svr_stat_svr_meters** struct.

os_svr_stat_svr_parameters_delete

```
void os_svr_stat_svr_parameters_delete(
        os_svr_stat_svr_parameters* /* svr_param */,
        os_boolean /* is_array */
);
```

Deletes an **os_svr_stat_svr_parameters** struct.

os_svr_stat_svr_rusage_delete

```
void os_svr_stat_svr_rusage_delete(
        os_svr_stat_svr_rusage* /* svr_resources */,
        os_boolean /* is_array */
);
```

Deletes an **os_svr_stat_svr_rusage** struct.

os_verifydb_options_allocate

os_verifydb_options* os_verifydb_options_allocate

Allocates an appropriately initialized **os_verifydb_options** struct. See **os_dbutil::osverifydb()** on page 114.

os_verifydb_options_delete

```
void os_verifydb_options_delete(
        os_verifydb_options* opts
);
```

Deletes an **os_verifydb_options** struct.

os_keyword_arg Functions

The C library interface contains functions analogous to those of the class **os_keyword_arg** in the ObjectStore class library.

os_keyword_arg_char_create

```
extern os_keyword_arg* os_keyword_arg_char_create(
    char *,         /* the name of the keyword argument */
    char           /* the value */
);
```

Creates a keyword argument. See **os_keyword_arg::operator ,()** and **os_keyword_arg::os_keyword_arg()** in *ObjectStore Collections C++ API Reference*.

os_keyword_arg_uchar_create

```
extern os_keyword_arg* os_keyword_arg_uchar_create(
    char *,        /* the name of the keyword argument */
    unsigned char/* the value */
);
```

Creates a keyword argument. See **os_keyword_arg::operator ,()** and **os_keyword_arg::os_keyword_arg()** in *ObjectStore Collections C++ API Reference*.

os_keyword_arg_short_create

```
extern os_keyword_arg* os_keyword_arg_short_create(
    char *,        /* the name of the keyword argument */
    short          /* the value */
);
```

Creates a keyword argument. See **os_keyword_arg::operator ,()** and **os_keyword_arg::os_keyword_arg()** in *ObjectStore Collections C++ API Reference*.

os_keyword_arg_ushort_create

```
extern os_keyword_arg* os_keyword_arg_ushort_create(
    char *,        /* the name of the keyword argument */
    unsigned short/* the value */
);
```

Creates a keyword argument. See **os_keyword_arg::operator ,()** and **os_keyword_arg::os_keyword_arg()** in *ObjectStore Collections C++ API Reference*.

os_keyword_arg_int_create

```
extern os_keyword_arg* os_keyword_arg_int_create(
    char *,       /* the name of the keyword argument */
    int           /* the value */
);
```

Creates a keyword argument. See **os_keyword_arg::operator ,()** and **os_keyword_arg::os_keyword_arg()** in *ObjectStore Collections C++ API Reference*.

os_keyword_arg_uint_create

```
extern os_keyword_arg* os_keyword_arg_uint_create(
    char *,       /* the name of the keyword argument */
    unsigned int/* the value */
);
```

Creates a keyword argument. See **os_keyword_arg::operator ,()** and **os_keyword_arg::os_keyword_arg()** in *ObjectStore Collections C++ API Reference*.

os_keyword_arg_long_create

```
extern os_keyword_arg* os_keyword_arg_long_create(
    char *,       /* the name of the keyword argument */
    long          /* the value */
);
```

Creates a keyword argument. See **os_keyword_arg::operator ,()** and **os_keyword_arg::os_keyword_arg()** in *ObjectStore Collections C++ API Reference*.

os_keyword_arg_ulong_create

```
extern os_keyword_arg* os_keyword_arg_ulong_create(
    char *,       /* the name of the keyword argument */
    unsigned long/* the value */
);
```

Creates a keyword argument. See **os_keyword_arg::operator ,()** and **os_keyword_arg::os_keyword_arg()** in *ObjectStore Collections C++ API Reference*.

os_keyword_arg_float_create

```
extern os_keyword_arg* os_keyword_arg_float_create(
    char *,       /* the name of the keyword argument */
    float         /* the value */
);
```

Creates a keyword argument. See **os_keyword_arg::operator ,()** and **os_keyword_arg::os_keyword_arg()** in *ObjectStore Collections C++ API Reference*.

os_keyword_arg_double_create

```
extern os_keyword_arg* os_keyword_arg_double_create(
    char *,       /* the name of the keyword argument */
    double        /* the value */
);
```

Creates a keyword argument. See **os_keyword_arg::operator ,()** and **os_keyword_arg::os_keyword_arg()** in *ObjectStore Collections C++ API Reference*.

os_keyword_arg_long_double_create

```
extern os_keyword_arg* os_keyword_arg_long_double_create(
    char *,       /* the name of the keyword argument */
    long double   /* the value */
);
```

Creates a keyword argument. See **os_keyword_arg::operator ,()** and **os_keyword_arg::os_keyword_arg()** in *ObjectStore Collections C++ API Reference*.

os_keyword_arg_pointer_create

```
extern os_keyword_arg* os_keyword_arg_pointer_create(
    char *,       /* the name of the keyword argument */
    void *        /* the value */
);
```

Creates a keyword argument. See **os_keyword_arg::operator ,()** and **os_keyword_arg::os_keyword_arg()** in *ObjectStore Collections C++ API Reference*.

os_keyword_arg_list Functions

The C library interface contains functions analogous to those of the class **os_keyword_arg_list** in the ObjectStore Class Library.

os_keyword_arg_list_create

```
extern os_keyword_arg_list* os_keyword_arg_list_create(
    os_keyword_arg*, /* arg to put at head of list */
    os_keyword_arg_list*/* tail (or 0) */
);
```

Creates a keyword argument list. See **os_keyword_arg::operator ,()** and **os_keyword_arg::os_keyword_arg()** in *ObjectStore Collections C++ API Reference*.

os_keyword_arg_list_delete

```
extern void os_keyword_arg_list_delete(
    os_keyword_arg_list*
);
```

Deletes a keyword argument list.

os_lock_timeout_exception Functions

The C library interface contains functions analogous to those of the class **os_lock_timeout_exception** in the ObjectStore Class Library.

os_lock_timeout_exception_get_application_names

```
extern char ** os_lock_timeout_exception_get_application_names(
    os_lock_timeout_exception *
);
```

See **os_lock_timeout_exception::get_application_names()** on page 152.

os_lock_timeout_exception_get_fault_addr

```
extern void * os_lock_timeout_exception_get_fault_addr(
    os_lock_timeout_exception *
);
```

See **os_lock_timeout_exception::get_fault_addr()** on page 152.

os_lock_timeout_exception_get_hostnames

```
extern char ** os_lock_timeout_exception_get_hostnames(
    os_lock_timeout_exception *
);
```

See **os_lock_timeout_exception::get_hostnames()** on page 152.

os_lock_timeout_exception_get_lock_type

```
enum os_lock_type os_lock_timeout_exception_get_lock_type(
    os_lock_timeout_exception *
);
```

See **os_lock_timeout_exception::get_lock_type()** on page 153.

os_lock_timeout_exception_get_pids

```
extern os_pid * os_lock_timeout_exception_get_pids(
    os_lock_timeout_exception *
);
```

See **os_lock_timeout_exception::get_pids()** on page 153.

os_lock_timeout_exception_number_of_blockers

```
extern int os_lock_timeout_exception_number_of_blockers(
    os_lock_timeout_exception *
```

);

See **os_lock_timeout_exception::number_of_blockers()** on page 153.

Metaobject Protocol Functions

The C library interface contains functions analogous to those of the ObjectStore metaobject protocol.

os_fetch_address_os_base_class

```
extern void* os_fetch_address_os_base_class(
    void *,
    const os_base_class *
);
```

See **::os_fetch_address()** on page 373.

os_fetch_address_os_member_variable

```
extern void* os_fetch_address_os_member_variable(
    void *,
    const os_member_variable *
);
```

See **::os_fetch_address()** on page 373.

char_os_fetch

```
extern char char_os_fetch(
    const void *,
    const os_member_variable *,
    char */* char value being fetched */
);
```

See **::os_fetch()** on page 370.

char_os_store

```
extern void char_os_store(
    void *,
    const os_member_variable *,
    const char/* const char value to be stored */
);
```

See **::os_store()** on page 374.

double_os_fetch

```
extern double double_os_fetch(
    const void *,
    const os_member_variable *,
    double */* double value being fetched */
);
```

See **::os_store()** on page 374.

double_os_store

```
extern void double_os_store(
    void *,
    const os_member_variable *,
    const double/* double value to be stored */
);
```

See **::os_store()** on page 374.

float_os_fetch

```
extern float float_os_fetch(
    const void *,
    const os_member_variable *,
    float */* float value being fetched */
);
```

See **::os_fetch()** on page 370.

float_os_store

```
extern void float_os_store(
    void *,
    const os_member_variable *,
    const float/* float value to be stored */
);
```

See **::os_store()** on page 374.

int_os_fetch

```
extern int int_os_fetch(
    const void *,
    const os_member_variable *,
    int */* int value being fetched */
);
```

See **::os_fetch()** on page 370.

int_os_store

```
extern void int_os_store(
    void *,
    const os_member_variable *,
    const int/*const int value to be stored */
);
```

See **::os_store()** on page 374.

long_double_os_fetch

```
extern long double long_double_os_fetch(
    const void *,
    const os_member_variable *,
    long double */* double value being fetched */
);
```

See **::os_fetch()** on page 370.

long_double_os_store

```
extern void long_double_os_store(
    void *,
    const os_member_variable *,
    const long double/* double value to be stored */
);
```

See **::os_store()** on page 374.

long_os_fetch

```
extern long long_os_fetch(
    const void *,
    const os_member_variable *,
    long */* long value being fetched */
);
```

See **::os_fetch()** on page 370.

long_os_store

```
extern void long_os_store(
    void *,
    const os_member_variable *,
    const long/* long value to be stored */
);
```

See **::os_store()** on page 374.

pointer_to_void_os_fetch

```
extern void* pointer_to_void_os_fetch(
    const void *,
    const os_member_variable *,
    os_void_p*/* value being fetched */
);
```

See **::os_fetch()** on page 370.

pointer_to_void_os_store

```
extern void pointer_to_void_os_store(
    void *,
    const os_member_variable *,
    const void */* value to be stored */
);
```

See **::os_store()** on page 374.

short_os_fetch

```
extern short short_os_fetch(
    const void *,
    const os_member_variable *,
    short */* short value being fetched */
);
```

See **::os_fetch()** on page 370.

short_os_store

```
extern void short_os_store(
    void *,
    os_member_variable *,
    const short/* const short value to be stored */
);
```

See **::os_store()** on page 374.

signed_char_os_fetch

```
extern signed char signed_char_os_fetch(
    const void *,
    const os_member_variable *,
    signed char */* signed char value being fetched */
);
```

See **::os_fetch()** on page 370.

signed_char_os_store

```
extern void signed_char_os_store(
    void *,
    const os_member_variable *,
    const signed char/* signed char value to be stored */
);
```

See **::os_store()** on page 374.

unsigned_char_os_fetch

```
extern unsigned char unsigned_char_os_fetch(
    const void *,
    const os_member_variable *,
    unsigned char */* unsigned char value being fetched */
);
```

See **::os_fetch()** on page 370.

unsigned_char_os_store

```
extern void unsigned_char_os_store(
    void *,
    const os_member_variable *,
    const unsigned char/* unsigned char value to be stored */
);
```

See **::os_store()** on page 374.

unsigned_int_os_fetch

```
extern unsigned int unsigned_int_os_fetch(
    const void *,
    const os_member_variable *,
    unsigned int */* unsigned int value being fetched */
);
```

See **::os_fetch()** on page 370.

unsigned_int_os_store

```
extern void unsigned_int_os_store(
    void *,
    const os_member_variable *,
    const unsigned int/* const unsigned int value to be stored */
);
```

See **::os_store()** on page 374.

unsigned_long_os_fetch

```
extern unsigned long unsigned_long_os_fetch(
    const void *,
    const os_member_variable *,
    unsigned long */* unsigned long value being fetched */
);
```

See **::os_fetch()** on page 370.

unsigned_long_os_store

```
extern void unsigned_long_os_store(
    void *,
    const os_member_variable *,
    const unsigned long/* unsigned long value to be stored */
);
```

See **::os_store()** on page 374.

unsigned_short_os_fetch

```
short unsigned_short_os_fetch
extern unsigned short unsigned_short_os_fetch(
    const void *,
    const os_member_variable *,
    unsigned short */* unsigned short value being fetched */
);
```

See **::os_fetch()** on page 370.

unsigned_short_os_store

```
extern void unsigned_short_os_store(
    void *,
    const os_member_variable *,
    const unsigned short/* const unsigned short value to be stored */
);
```

See **::os_store()** on page 374.

wchar_t_os_fetch

```
extern wchar_t wchar_t_os_fetch(
    const void *,
    const os_member_variable *,
    wchar_t */* wchar_t value being fetched */
);
```

See **::os_fetch()** on page 370.

wchar_t_os_store

```
extern void wchar_t_os_store(
    void *,
    const os_member_variable *,
    const wchar_t/* const wchar_t value to be stored */
);
```

See **::os_store()** on page 374.

os_access_modifier_cast_to_os_member

```
extern os_member* os_access_modifier_cast_to_os_member(
    os_access_modifier *
);
```

Cast to **os_member**.

os_access_modifier_create

```
extern os_access_modifier* os_access_modifier_create(
    os_member* /*member*/
);
```

See **os_access_modifier::create()** on page 44.

os_access_modifier_get_base_member

```
extern os_member* os_access_modifier_get_base_member(
    os_access_modifier *
);
```

See **os_access_modifier::get_base_member()** on page 44.

os_access_modifier_set_base_member

```
extern void os_access_modifier_set_base_member(
    os_access_modifier* /*modifier*/,
    os_member* /*member*/
);
```

See **os_access_modifier::set_base_member()** on page 44.

os_anon_indir_type_cast_to_os_indirect_type

```
extern os_indirect_type*
os_anon_indir_type_cast_to_os_indirect_type(
    os_anonymous_indirect_type *
);
```

Cast to supertype.

os_anon_indir_type_cast_to_os_type

```
extern os_type* os_anon_indir_type_cast_to_os_type(
    os_anonymous_indirect_type *
);
```

Cast to supertype.

os_anon_indir_type_create

```
extern os_anonymous_indirect_type* os_anon_indir_type_create(
    os_type* /*target_type*/
);
```

See **os_anonymous_indirect_type::create()** on page 49.

os_anon_indir_type_get_target_type

```
extern os_type* os_anon_indir_type_get_target_type(
    os_anonymous_indirect_type *
);
```

See **os_anonymous_indirect_type::get_target_type()** on page 49.

os_anon_indir_type_is_const

```
extern os_boolean os_anon_indir_type_is_const(
    os_anonymous_indirect_type*
);
```

See **os_anonymous_indirect_type::is_const()** on page 49.

os_anon_indir_type_is_volatile

```
extern os_boolean os_anon_indir_type_is_volatile(
    os_anonymous_indirect_type*
);
```

See **os_anonymous_indirect_type::is_volatile()** on page 49.

os_anon_indir_type_set_is_const

```
extern void os_anon_indir_type_set_is_const(
    os_anonymous_indirect_type*,
    os_boolean
);
```

See **os_anonymous_indirect_type::set_is_const()** on page 49.

os_anon_indir_type_set_is_volatile

```
extern void os_anon_indir_type_set_is_volatile(
    os_anonymous_indirect_type*,
    os_boolean
);
```

See **os_anonymous_indirect_type::set_is_volatile()** on page 49.

os_app_schema_cast_to_os_schema

```
extern os_schema* os_app_schema_cast_to_os_schema(
    os_app_schema *
);
```

Cast to supertype.

os_app_schema_get

```
extern os_app_schema* os_app_schema_get(
    os_database *
    /* return the application schema stored in this database */
);
```

See **os_app_schema::get()** on page 50.

os_app_schema_get_default

```
extern os_app_schema* os_app_schema_get_default();
```

See **os_app_schema::get()** on page 50.

os_array_type_cast_to_os_type

```
extern os_type* os_array_type_cast_to_os_type(
    os_array_type *
);
```

Cast to supertype.

os_array_type_create

```
extern os_array_type* os_array_type_create(
    os_unsigned_int32 /*number_of_elements*/,
    os_type* /*element_type*/
);
```

See **os_array_type::create()** on page 52.

os_array_type_get_element_type

```
extern os_type* os_array_type_get_element_type(
    os_array_type *
);
```

See **os_array_type::get_element_type()** on page 52.

os_array_type_number_of_elements

```
extern os_unsigned_int32 os_array_type_number_of_elements(
    os_array_type *
);
```

See **os_array_type::get_number_of_elements()** on page 52.

os_array_type_set_element_type

```
extern void os_array_type_set_element_type(
    os_array_type* /*array_type*/,
    os_type* /*element_type*/
);
```

See **os_array_type::set_element_type()** on page 52.

os_array_type_set_number_of_elements

```
extern void os_array_type_set_number_of_elements(
    os_array_type* /*array_type*/,
    os_unsigned_int32 /*number_of_elements*/
);
```

See **os_array_type::set_number_of_elements()** on page 53.

os_base_class_create

```
extern os_base_class* os_base_class_create(
    os_unsigned_int32 /*access*/,
    os_boolean /*is_virtual*/,
    os_class_type* /*class_type*/
);
```

See **os_base_class::create()** on page 54.

os_base_class_get_access

```
extern os_base_class_access os_base_class_get_access(
    os_base_class *
);
```

See **os_base_class::get_access()** on page 54.

os_base_class_get_class

```
extern os_class_type* os_base_class_get_class(
    os_base_class *
);
```

See **os_base_class::get_class()** on page 54.

os_base_class_get_offset

```
extern os_unsigned_int32 os_base_class_get_offset(
    os_base_class *
);
```

See **os_base_class::get_offset()** on page 55.

os_base_class_get_offset_in

```
extern os_unsigned_int32 os_base_class_get_offset_in(
    os_base_class* /*virtual_base*/,
    os_class_type* /*most_derived_class*/
);
```

See **os_base_class::get_offset()** on page 55.

os_base_class_get_size

```
extern os_unsigned_int32 os_base_class_get_size(
    os_base_class *
);
```

See **os_base_class::get_size()** on page 55.

os_base_class_get_virtual_base_class_pointer_offset

```
extern os_unsigned_int32
os_base_class_get_virtual_base_class_pointer_offset(
    os_base_class* /*base*/
    ) /*throw (err_mop)*/
);
```

See **os_base_class::get_virtual_base_class_pointer_offset()** on page 55.

os_base_class_is_virtual

```
extern os_boolean os_base_class_is_virtual(
    os_base_class */* return true if this is a virtual */
             /* base class definition*/
);
```

See **os_base_class::is_virtual()** on page 55.

os_base_class_set_access

```
extern void os_base_class_set_access(
    os_base_class* /*base*/,
    os_unsigned_int32 /*os_base_access*/
);
```

See **os_base_class::set_access()** on page 55.

os_base_class_set_class

```
extern void os_base_class_set_class(
    os_base_class* /*base*/,
    os_class_type* /*class_type*/
);
```

See **os_base_class::set_class()** on page 56.

os_base_class_set_is_virtual

```
extern void os_base_class_set_is_virtual(
    os_base_class* /*base*/,
    os_boolean /*is_virtual*/
);
```

See **os_base_class::is_virtual()** on page 55.

os_base_class_set_offset

```
extern void os_base_class_set_offset(
    os_base_class* /*base*/,
    os_unsigned_int32 /*offset*/
);
```

See **os_base_class::set_offset()** on page 56.

os_base_class_set_virtual_base_class_no_pointer

```
extern void os_base_class_set_virtual_base_class_no_pointer(
    os_base_class* /*base*/
    ) /*throw (err_mop)*/;
```

See **os_base_class::set_virtual_base_class_no_pointer()** on page 56.

os_base_class_set_virtual_base_class_pointer_offset

```
extern void os_base_class_set_virtual_base_class_pointer_offset(
    os_base_class* /*base*/,
    os_unsigned_int32 offset
    ) /*throw (err_mop)*/;
```

See **os_base_class::set_virtual_base_class_pointer_offset()** on page 56.

os_base_class_set_virtuals_redefined

```
extern void os_base_class_set_virtuals_redefined(
    os_base_class* /*base*/,
    os_boolean /*virtuals_redefined*/
);
```

See **os_base_class::set_virtuals_redefined()** on page 56.

os_base_class_virtual_base_class_has_pointer

```
extern os_boolean os_base_class_virtual_base_class_has_
pointer(
    os_base_class* /*base*/
    ) /*throw (err_mop)*/;
```

See **os_base_class::virtual_base_class_has_pointer()** on page 56.

os_base_class_virtuals_redefined

```
extern os_boolean os_base_class_virtuals_redefined(
    os_base_class*
);
```

See **os_base_class::virtuals_redefined()** on page 56.

os_class_type_cast_to_os_instantiated_class_type

```
extern os_instantiated_class_type*
os_class_type_cast_to_os_instantiated_class_type(
    os_class_type *
);
```

See **os_class_type::operator os_instantiated_class_type&()** on page 66.

os_class_type_cast_to_os_type

```
extern os_type* os_class_type_cast_to_os_type(
    os_class_type *
);
```

Cast to supertype.

os_class_type_create

```
extern os_class_type* os_class_type_create(
    char* /*name*/,
    os_collection* /*base_classes*/,
    os_collection* /*members*/,
    os_boolean /*defines_virtual_functions*/
);
```

See **os_class_type::create()** on page 57.

os_class_type_create_forward_definition

```
extern os_class_type* os_class_type_create_forward_definition(
    char* /*name*/
);
```

See **os_class_type::create()** on page 57.

os_class_type_declares_get_os_typespec_function

```
extern os_boolean
os_class_type_declares_get_os_typespec_function(
    os_class_type* /*class_type*/
);
```

See **os_class_type::declares_get_os_typespec_function()** on page 58.

os_class_type_defines_virtual_functions

```
extern os_boolean os_class_type_defines_virtual_functions(
    os_class_type* /*class_type*/
);
```

See **os_class_type::defines_virtual_functions()** on page 58.

os_class_type_find_base_class

```
extern os_base_class* os_class_type_find_base_class(
    os_class_type *,
    const char */* name of potential direct base class */
);
```

See **os_class_type::find_base_class()** on page 58.

os_class_type_find_member

```
extern os_member* os_class_type_find_member(
    os_class_type *,
    const char *
);
```

See **os_class_type::find_member_variable()** on page 59.

os_class_type_get_access_of_get_os_typespec_function

```
extern os_mop_member_access
os_class_type_get_access_of_get_os_typespec_function(
    os_class_type* /*class_type*/
);
```

See **os_class_type::get_access_of_get_os_typespec_function()** on page 59.

os_class_type_get_allocated_virtual_base_classes

```
extern os_collection*os_class_type_get_allocated_virtual_base_
classes(
    os_class_type *
);
```

See **os_class_type::get_allocated_virtual_base_classes()** on page 59.

os_class_type_get_base_classes

```
extern os_collection* os_class_type_get_base_classes(
    os_class_type *
);
```

See **os_class_type::get_base_classes()** on page 60.

os_class_type_get_class_kind

```
extern os_class_type_kind os_class_type_get_class_kind(
    os_class_type*
);
```

See **os_class_type::get_class_kind()** on page 60.

os_class_type_get_dispatch_table_pointer_offset

```
extern os_int32 os_class_type_get_dispatch_table_pointer_offset(
    os_class_type* /*class_type*/
    ) /*throw(err_mop_forward_definition)*/;
```

See **os_class_type::get_dispatch_table_pointer_offset()** on page 60.

os_class_type_get_indirect_virtual_base_classes

```
extern os_collection*
os_class_type_get_indirect_virtual_base_classes(
    os_class_type* /*class_type*/
    ) /*throw(err_mop_forward_definition)*/;
```

See **os_class_type::get_indirect_virtual_base_classes()** on page 61.

os_class_type_get_members

```
extern os_collection* os_class_type_get_members(
    os_class_type *
);
```

See **os_class_type::get_members()** on page 61.

os_class_type_get_most_derived_class

```
extern os_class_type* os_class_type_get_most_derived_class(
    /*const*/ void* /*object*/,
    /*const*/ os_void_p* /*most_derived_object*/
);
```

See **os_class_type::get_most_derived_class()** on page 61.

os_class_type_get_name

```
extern char* os_class_type_get_name(
    os_class_type *
);
```

See **os_class_type::get_name()** on page 64.

os_class_type_get_pragmas

```
extern os_collection* os_class_type_get_pragmas(
    os_class_type *
);
```

See **os_class_type::get_pragmas()** on page 64.

os_class_type_get_size_as_base

```
extern os_unsigned_int32 os_class_type_get_size_as_base(
    os_class_type *
);
```

See **os_class_type::get_size_as_base()** on page 64.

os_class_type_get_source_position

```
extern void os_class_type_get_source_position(
    os_class_type* /*class_type*/,
    os_char_p* /*file*/,
    os_unsigned_int32* /*line*/
);
```

See **os_class_type::get_source_position()** on page 64.

os_class_type_has_constructor

```
extern os_boolean os_class_type_has_constructor(os_class_
type*);
```

See **os_class_type::has_constructor()** on page 65.

os_class_type_has_destructor

```
extern os_boolean os_class_type_has_destructor(os_class_type*);
```

See **os_class_type::has_destructor()** on page 65.

os_class_type_has_dispatch_table

```
extern os_boolean os_class_type_has_dispatch_table(
    os_class_type* /*class_type*/
    ) /*throw (err_mop_forward_definition)*/;
```

See **os_class_type::has_dispatch_table()** on page 65.

os_class_type_introduces_virtual_functions

```
extern os_boolean os_class_type_introduces_virtual_functions(
    os_class_type* /*class_type*/
);
```

See **os_class_type::introduces_virtual_functions()** on page 65.

os_class_type_is_abstract

```
extern os_boolean os_class_type_is_abstract(
    os_class_type *
);
```

See **os_class_type::is_abstract()** on page 65.

os_class_type_is_forward_definition

```
extern os_boolean os_class_type_is_forward_definition(
    os_class_type *
);
```

See **os_class_type::is_forward_definition()** on page 65.

os_class_type_is_persistent

```
extern os_boolean os_class_type_is_persistent(
    os_class_type *
);
```

See **os_class_type::is_persistent()** on page 65.

os_class_type_is_template_class

```
extern os_boolean os_class_type_is_template_class(
    os_class_type *
);
```

See **os_class_type::is_template_class()** on page 66.

os_class_type_set_access_of_get_os_typespec_function

```
extern void
os_class_type_set_access_of_get_os_typespec_function(
    os_class_type* /*class_type*/,
    os_unsigned_int32 /*access*/
);
```

See **os_class_type::set_access_of_get_os_typespec_function()** on page 66.

os_class_type_set_base_classes

```
extern void os_class_type_set_base_classes(
    os_class_type* /*class_type*/,
    os_collection* /*base_classes*/
);
```

See **os_class_type::set_base_classes()** on page 66.

os_class_type_set_class_kind

```
extern void os_class_type_set_class_kind(
    os_class_type* /*class_type*/,
    os_unsigned_int32 /*class_kind*/
);
```

See **os_class_type::set_class_kind()** on page 66.

os_class_type_set_defines_virtual_functions

```
extern void os_class_type_set_defines_virtual_functions(
    os_class_type* /*class_type*/,
    os_boolean /*defines_virtual_functions*/
);
```

See **os_class_type::set_defines_virtual_functions()** on page 67.

os_class_type_set_dispatch_table_pointer_offset

```
extern void os_class_type_set_dispatch_table_pointer_offset(
    os_class_type* /*class_type*/,
    os_int32 /*offset*/
);
```

See **os_class_type::set_dispatch_table_pointer_offset()** on page 67.

os_class_type_set_has_constructor

```
extern os_boolean
 os_class_type_set_has_constructor(os_class_type*, os_boolean);
```

See **os_class_type::set_has_constructor()** on page 67.

os_class_type_set_has_destructor

```
extern os_boolean
   os_class_type_set_has_destructor(os_class_type*, os_boolean);
```

See **os_class_type::set_has_destructor()** on page 67.

os_class_type_set_declares_get_os_typespec_function

```
extern void    os_class_type_set_declares_get_os_typespec_
function(
    os_class_type* /*class_type*/,
    os_boolean /*declares_get_os_typespec_function*/
);
```

See **os_class_type::set_declares_get_os_typespec_function()** on page 67.

os_class_type_set_indirect_virtual_base_classes

```
extern void os_class_type_set_indirect_virtual_base_classes(
    os_class_type* /*class_type*/,
    os_collection* /*base_classes*/
);
```

See **os_class_type::set_indirect_virtual_base_classes()** on page 67.

os_class_type_set_introduces_virtual_functions

```
extern void os_class_type_set_introduces_virtual_functions(
    os_class_type* /*class_type*/,
    os_boolean /*defines_new_virtual_functions*/
);
```

See **os_class_type::set_introduces_virtual_functions()** on page 67.

os_class_type_set_is_abstract

```
extern void os_class_type_set_is_abstract(
    os_class_type* /*class_type*/,
    os_boolean /*is_abstract*/
);
```

See **os_class_type::set_is_abstract()** on page 67.

os_class_type_set_is_forward_definition

```
extern void os_class_type_set_is_forward_definition(
    os_class_type* /*class_type*/,
    os_boolean /*is_forward*/
);
```

See **os_class_type::set_is_forward_definition()** on page 68.

os_class_type_set_is_persistent

```
extern void os_class_type_set_is_persistent(
    os_class_type* /*class_type*/,
    os_boolean /*is_persistent*/
);
```

See **os_class_type::set_is_persistent()** on page 68.

os_class_type_set_members

```
extern void os_class_type_set_members(
    os_class_type* /*class_type*/,
    os_collection* /*members*/
);
```

See **os_class_type::set_members()** on page 68.

os_class_type_set_name

```
extern void os_class_type_set_name(
    os_class_type* /*class_type*/,
    char* /*name*/
);
```

See **os_class_type::set_name()** on page 68.

os_class_type_set_pragmas

```
extern void os_class_type_set_pragmas(
    os_class_type* /*class_type*/,
    os_collection* /*pragmas*/
);
```

See **os_class_type::set_pragmas()** on page 68.

os_class_type_set_source_position

```
extern void os_class_type_set_source_position(
    os_class_type* /*class_type*/,
    /*const*/ char* /*file*/,
    /*const*/ os_unsigned_int32 /*line*/
);
```

See **os_class_type::set_source_position()** on page 68.

os_comp_schema_cast_to_os_schema

```
extern os_schema* os_comp_schema_cast_to_os_schema(
    os_comp_schema *
);
```

Cast to supertype.

os_comp_schema_get

```
extern os_comp_schema* os_comp_schema_get(
    os_database *
    /* return the compilation schema stored in this database */
);
```

See **os_comp_schema::get()** on page 70.

os_database_schema_cast_to_os_schema

```
extern os_schema* os_database_schema_cast_to_os_schema(
    os_database_schema *
);
```

Cast to supertype.

os_database_schema_get

```
extern os_database_schema* os_database_schema_get(
    os_database *
    /* return the schema associated with this database */
);
```

See **os_database_schema::get()** on page 105.

os_database_schema_get_for_update

```
extern os_database_schema* os_database_schema*
os_database_schema_get_for_update(
    /*const*/ os_database*
    ) /*throw(err_no_schema)*/;
```

See **os_database_schema::get_for_update()** on page 105.

os_database_schema_install

```
extern void os_database_schema_install(
    os_database_schema* /*target_schema*/,
    os_schema* /*new_schema*/
    ) /*throw (err_schema_validation_error)*/;
```

See **os_database_schema::install()** on page 105.

os_database_schema_install_with_options

```
extern void os_database_schema_install_with_options(
    os_database_schema*,
        os_schema*,
        os_schema_install_options*
);
```

See **os_database_schema::install()** on page 105.

os_enum_type_cast_to_os_type

```
extern os_type* os_enum_type_cast_to_os_type(
    os_enum_type *
);
```

Cast to supertype.

os_enum_type_create

```
extern os_enum_type* os_enum_type_create(
    char* /*name*/,
    os_collection* /*literals*/
);
```

See **os_enum_type::create()** on page 135.

os_enum_type_get_enumerator

```
extern os_enumerator_literal* os_enum_type_get_enumerator(
    os_enum_type *,
    os_int32
);
```

See **os_enum_type::get_enumerator()** on page 135.

os_enum_type_get_enumerators

```
extern os_collection* os_enum_type_get_enumerators(
    os_enum_type *
);
```

See **os_enum_type::get_enumerators()** on page 135.

os_enum_type_get_name

```
extern char* os_enum_type_get_name(
    os_enum_type *
);
```

See **os_enum_type::get_name()** on page 135.

os_enum_type_get_pragmas

```
extern os_collection* os_enum_type_get_pragmas(
    os_enum_type *
);
```

See **os_enum_type::get_pragmas()** on page 136.

os_enum_type_get_source_position

```
extern void os_enum_type_get_source_position(
    os_enum_type*,
    os_char_p* /*file*/,
    os_unsigned_int32* /*line*/
);
```

See **os_enum_type::get_source_position()** on page 136.

os_enum_type_set_enumerators

```
extern void os_enum_type_set_enumerators(
    os_enum_type*,
    os_collection* /*literals*/
    ) /*throw (err_mop)*/;
```

See **os_enum_type::set_enumerators()** on page 136.

os_enum_type_set_name

```
extern void os_enum_type_set_name(
    os_enum_type*,
    char* /*name*/
);
```

See **os_enum_type::set_name()** on page 136.

os_enum_type_set_pragmas

```
extern void os_enum_type_set_pragmas(
    os_enum_type* /*enum_type*/,
    os_collection* /*pragmas*/
);
```

See **os_enum_type::set_pragmas()** on page 136.

os_enum_type_set_source_position

```
extern void os_enum_type_set_source_position(
    os_enum_type*,
    /*const*/ char* /*file*/,
    /*const*/ os_unsigned_int32 /*line*/
);
```

See **os_enum_type::set_source_position()** on page 136.

os_enumerator_literal_create

```
extern os_enumerator_literal* os_enumerator_literal_create(
    char* /*name*/,
    os_int32 /*value*/
);
```

See **os_enumerator_literal::create()** on page 137.

os_enumerator_literal_get_name

```
extern char* os_enumerator_literal_get_name(
    os_enumerator_literal *
);
```

See **os_enumerator_literal::get_name()** on page 137.

os_enumerator_literal_get_value

```
extern os_int32 os_enumerator_literal_get_value(
    os_enumerator_literal *
);
```

See **os_enumerator_literal::get_value()** on page 137.

os_enumerator_literal_set_name

```
extern void os_enumerator_literal_set_name(
    os_enumerator_literal* /*literal*/,
    char* /*name*/
);
```

See **os_enumerator_literal::set name()** on page 137.

os_enumerator_literal_set_value

```
extern void os_enumerator_literal_set_value(
    os_enumerator_literal* /*literal*/,
    os_int32 /*value*/
);
```

See **os_enumerator_literal::set_value()** on page 137.

os_field_member_variable_create

```
extern os_field_member_variable*
os_field_member_variable_create(
    char* /*name*/,
    os_integral_type* /*type*/,
    os_unsigned_int8 /*size_in_bits*/
);
```

See **os_field_member_variable::create()** on page 141.

os_field_member_variable_get_offset

```
extern void os_field_member_variable_get_offset(
    os_field_member_variable *,
    os_unsigned_int32 *,/* byte offset part */
    os_unsigned_int8 */* bit offset part*/
);
```

See **os_field_member_variable::get_offset()** on page 141.

os_field_member_variable_get_size

```
extern os_unsigned_int8 os_field_member_variable_get_size(
    os_field_member_variable */* return the size in bits */
                    /* occupied by this member */
);
```

See **os_field_member_variable::get_size()** on page 141.

os_field_member_variable_set_offset

```
extern void os_field_member_variable_set_offset(
    os_field_member_variable* /*field*/,
    os_unsigned_int32 /*bytes*/,
    os_unsigned_int8 /*bits*/
);
```

See **os_field_member_variable::set_offset()**.

os_field_member_variable_set_size

```
extern void os_field_member_variable_set_size(
    os_field_member_variable* /*field*/,
    os_unsigned_int8 /*size_in_bits*/
);
```

See **os_field_member_variable::set_size()** on page 141.

os_fld_mem_var_cast_to_os_mem_variable

```
extern os_member_variable*
os_fld_mem_var_cast_to_os_mem_variable(
    os_field_member_variable *
);
```

Cast to supertype.

os_fld_mem_var_cast_to_os_member

```
extern os_member* os_fld_mem_var_cast_to_os_member(
    os_field_member_variable *
);
```

Cast to supertype.

os_function_type_cast_to_os_type

```
extern os_type* os_function_type_cast_to_os_type(
    os_function_type *
);
```

Cast to supertype.

os_function_type_create

```
extern os_function_type* os_function_type_create(
    os_unsigned_int32 /*os_arg_list_kind*/,
    os_collection* /*args*/,
    os_type* /*return_type*/
);
```

See **os_function_type::create()** on page 142.

os_function_type_equal_signature

```
extern os_boolean os_function_type_equal_signature(
    os_function_type* /*function_type*/,
    os_function_type* /*other_function_type*/,
    os_boolean /*check_return_type*/
);
```

See **os_function_type::equal_signature()** on page 142.

os_function_type_get_arg_list

```
extern os_collection* os_function_type_get_arg_list(
    os_function_type *
);
```

See **os_function_type::get_arg_list()** on page 142.

os_function_type_get_arg_list_kind

```
extern os_function_arg_list_kindos_function_type_get_arg_list_
kind(
    os_function_type *
);
```

See **os_function_type::get_arg_list_kind()** on page 143.

os_function_type_get_return_type

```
extern os_type* os_function_type_get_return_type(
    os_function_type *
);
```

See **os_function_type::get_return_type()** on page 143.

os_function_type_set_arg_list

```
extern void os_function_type_set_arg_list(
    os_function_type* /*function_type*/,
    os_collection* /*args*/
);
```

See **os_function_type::set_arg_list()** on page 143.

os_function_type_set_arg_list_kind

```
extern void os_function_type_set_arg_list_kind(
    os_function_type* /*function_type*/,
    os_unsigned_int32 /*os_arg_list_kind*/
);
```

See **os_function_type::set_arg_list_kind()** on page 143.

os_function_type_set_return_type

```
extern void os_function_type_set_return_type(
    os_function_type* /*function_type*/,
    os_type* /*return_type*/
);
```

See **os_function_type::set_return_type()** on page 143.

os_indir_type_cast_to_os_anon_indir_type

```
extern os_anonymous_indirect_type*
os_indir_type_cast_to_os_anon_indir_type(
    os_indirect_type *
);
```

See **os_indirect_type** conversion operators.

os_indir_type_cast_to_os_named_indir_type

```
extern os_named_indirect_type*
os_indir_type_cast_to_os_named_indir_type(
    os_indirect_type *
);
```

See **os_indirect_type** conversion operators.

os_indir_type_get_target_type

```
extern os_type* os_indir_type_get_target_type(
    os_indirect_type *
);
```

See **os_indirect_type::get_target_type()** on page 144.

os_indir_type_set_target_type

```
extern void os_indir_type_set_target_type(
    os_indirect_type*,
    os_type* /*target_type*/
);
```

See **os_indirect_type::set_target_type()** on page 144.

os_indirect_type_cast_to_os_type

```
extern os_type* os_indirect_type_cast_to_os_type(
    os_indirect_type *
);
```

Cast to supertype.

os_inst_class_type_cast_to_os_class_type

```
extern os_class_type* os_inst_class_type_cast_to_os_class_type(
    os_instantiated_class_type *
);
```

Cast to supertype.

os_inst_class_type_cast_to_os_type

```
extern os_type* os_inst_class_type_cast_to_os_type(
    os_instantiated_class_type *
);
```

Cast to supertype.

os_instantiated_class_type_create_forward_definition

```
extern os_instantiated_class_type*
os_instantiate_class_type_create_forward_definition(
    char* /*name*/
);
```

See **os_instantiated_class_type::create()** on page 145.

os_instantiated_class_type_create

```
extern os_instantiated_class_type*
os_instantiated_class_type_create(
    char* /*name*/,
    os_collection* /*base_classes*/,
    os_collection* /*members*/,
    os_template_instantiation* /*instantiation*/,
    os_boolean /*defines_virtual_functions*/
);
```

See **os_instantiated_class_type::create()** on page 145.

os_instantiated_class_type_get_instantiation

```
os_template_instantiation
extern os_template_instantiation*
os_instantiated_class_type_get_instantiation(
    os_instantiated_class_type *
);
```

See **os_instantiated_class_type::get_instantiation()** on page 145.

os_instantiated_class_type_set_instantiation

```
extern void
os_instantiated_class_type_set_instantiation(
    os_instantiated_class_type* /*class_type*/,
    os_template_instantiation* /*instantiation*/
);
```

See **os_instantiated_class_type::set_instantiation()** on page 145.

os_integral_type_cast_to_os_type

```
extern os_type* os_integral_type_cast_to_os_type(
    os_integral_type *
);
```

Cast to supertype.

os_integral_type_create

```
extern os_integral_type* os_integral_type_create(
    const char*
);
```

See **os_integral_type::create()** on page 146.

os_integral_type_create_defaulted_char

```
extern os_integral_type* os_integral_type_create_defaulted_char(
    os_boolean /*sign*/
);
```

See **os_integral_type::create_defaulted_char()** on page 146.

os_integral_type_create_int

```
extern os_integral_type* os_integral_type_create_int(
    os_boolean /*sign*/
);
```

See **os_integral_type::create_int()** on page 146.

os_integral_type_create_long

```
extern os_integral_type* os_integral_type_create_long(
    os_boolean /*sign*/
);
```

See **os_integral_type::create_long()** on page 146.

os_integral_type_create_short

```
extern os_integral_type* os_integral_type_create_short(
    os_boolean /*sign*/
);
```

See **os_integral_type::create_short()** on page 146.

os_integral_type_create_signed_char

```
extern os_integral_type* os_integral_type_create_signed_char();
```

See **os_integral_type::create_signed_char()** on page 147.

os_integral_type_create_unsigned_char

```
extern os_integral_type* os_integral_type_create_unsigned_char();
```

See **os_integral_type::create_unsigned_char()** on page 147.

os_integral_type_create_wchar_t

```
extern os_integral_type* os_integral_type_create_wchar_t();
```

See **os_integral_type::create_wchar_t()**.

os_integral_type_is_signed

```
extern os_boolean os_integral_type_is_signed(
    os_integral_type *
);
```

See **os_integral_type::is_signed()** on page 147.

os_literal_create_char

```
extern os_literal* os_literal_create_char(
    char /*c*/
);
```

See **os_literal::create_char()** on page 148.

os_literal_create_enum_literal

```
extern os_literal* os_literal_create_enum_literal(
    os_enumerator_literal*
);
```

See **os_literal::create_enum_literal()** on page 148.

os_literal_create_pointer_literal

```
extern os_literal* os_literal_create_pointer_literal(
    os_pointer_literal*
);
```

See **os_literal::create_pointer_literal()** on page 148.

os_literal_create_signed_char

```
extern os_literal* os_literal_create_signed_char(
    signed char
);
```

See **os_literal::create_signed_char()** on page 148.

os_literal_create_signed_int

```
extern os_literal* os_literal_create_signed_int(
    signed int
);
```

See **os_literal::create_signed_int()** on page 148.

os_literal_create_signed_long

```
extern os_literal* os_literal_create_signed_long(
    signed long
);
```

See **os_literal::create_signed_long()** on page 148.

os_literal_create_signed_int

```
extern os_literal* os_literal_create_signed_short(
    signed short
);
```

See **os_literal::create_signed_short()** on page 148.

os_literal_create_unsigned_char

```
extern os_literal* os_literal_create_unsigned_char(
    unsigned char
);
```

See **os_literal::create_unsigned_char()** on page 149.

os_literal_create_unsigned_int

```
extern os_literal* os_literal_create_unsigned_int(
    unsigned int
);
```

See **os_literal::create_unsigned_int()** on page 149.

os_literal_create_unsigned_long

```
extern os_literal* os_literal_create_unsigned_long(
    unsigned long
);
```

See **os_literal::create_unsigned_long()** on page 149.

os_literal_create_unsigned_short

```
extern os_literal* os_literal_create_unsigned_short(
    unsigned short
);
```

See **os_literal::create_unsigned_short()** on page 149.

os_literal_create_wchar_t

```
extern os_literal* os_literal_create_wchar_t(
    wchar_t /*c*/
);
```

See **os_literal::create_wchar_t()** on page 149.

os_literal_get_char_value

```
extern char os_literal_get_char_value(
    os_literal* /*literal*/
);
```

See **os_literal::get_char_value()** on page 149.

os_literal_get_enum_literal

```
extern os_enumerator_literal* os_literal_get_enum_literal(
    os_literal* /*literal*/
);
```

See **os_literal::get_enum_literal()** on page 149.

os_literal_get_kind

```
extern os_mop_literal_kind os_literal_get_kind(
    os_literal* /*literal*/
);
```

See **os_literal::get_kind()** on page 149.

os_literal_get_pointer_literal

```
extern os_pointer_literal* os_literal_get_pointer_literal(
    os_literal* /*literal*/
);
```

See **os_literal::get_pointer_literal()** on page 150.

os_literal_get_signed_integral_value

```
extern long os_literal_get_signed_integral_value(
    os_literal* /*literal*/
);
```

See **os_literal::get_signed_integral_value()** on page 150.

os_literal_get_type

```
extern os_type* os_literal_get_type(
    os_literal* /*literal*/
);
```

See **os_literal::get_type()** on page 150.

os_literal_get_unsigned_integral_value

```
extern unsigned long os_literal_get_unsigned_integral_value(
    os_literal* /*literal*/
);
```

See **os_literal::get_unsigned_integral_value()** on page 150.

os_literal_get_wchar_t_value

```
extern wchar_t os_literal_get_wchar_t_value(
    os_literal* /*literal*/
);
```

See **os_literal::get_wchar_t_value()** on page 150.

os_literal_is_unspecified

```
extern os_boolean os_literal_is_unspecified(
    os_literal* /*literal*/
);
```

See **os_literal::is_unspecified()**.

os_literal_template_actual_arg

```
extern os_literal_template_actual_arg*
os_literal_template_actual_arg_create(
    os_literal* /*literal*/
);
```

See **os_literal_template_actual_arg::create()** on page 151.

os_literal_template_actual_arg_get_literal

```
extern os_literal* os_literal_template_actual_arg_get_literal(
    os_literal_template_actual_arg* /*arg*/
);
```

See **os_literal_template_actual_arg::get_literal()** on page 151.

os_literal_template_actual_arg_set_literal

```
extern void os_literal_template_actual_arg_set_literal(
    os_literal_template_actual_arg* /*arg*/,
    os_literal* /*literal*/
);
```

See **os_literal_template_actual_arg::set_literal()** on page 151.

os_mem_function_cast_to_os_member

```
extern os_member* os_mem_function_cast_to_os_member(
    os_member_function *
);
```

Cast to supertype.

os_mem_var_cast_to_os_field_member_variable

```
extern os_field_member_variable*
os_mem_var_cast_to_os_field_member_variable(
    os_member_variable *
);
```

See **os_member_variable::operator os_field_member_variable&()** on page 168.

os_mem_var_cast_to_os_member

```
extern os_member* os_mem_var_cast_to_os_member(
    os_member_variable *
);
```

Cast to supertype.

os_mem_var_cast_to_os_relationship_member_variable

```
extern os_relationship_member_variable*
os_mem_var_cast_to_os_relationship_member_variable(
    os_member_variable *
);
```

See **os_member_variable::operator os_relationship_member_ variable&()** on page 168.

os_member_cast_to_os_access_modifier

```
extern os_access_modifier*
os_member_cast_to_os_access_modifier(
    os_member *
);
```

See **os_member::operator os_access_modifier&()** on page 156.

os_member_cast_to_os_field_member_variable

```
extern os_field_member_variable*
os_member_cast_to_os_field_member_variable(
    os_member *
);
```

See **os_member::operator os_field_member_variable&()** on page 157.

os_member_cast_to_os_member_function

```
extern os_member_function*os_member_cast_to_os_member_
function(
    os_member *
);
```

See **os_member::operator os_member_function&()** on page 157.

os_member_cast_to_os_member_type

```
extern os_member_type* os_member_cast_to_os_member_type(
    os_member *
);
```

See **os_member::operator os_member_type&()** on page 157.

os_member_cast_to_os_member_variable

```
extern os_member_variable*
os_member_cast_to_os_member_variable(
    os_member *
);
```

See **os_member::operator os_member_variable&()** on page 157.

os_member_cast_to_os_relationship_member_variable

```
extern os_relationship_member_variable*
os_member_cast_to_os_relationship_member_variable(
    os_member *
);
```

See **os_member::operator os_relationship_member_variable&()** on page 157.

os_member_defining_class

```
extern os_class_type* os_member_defining_class(
    os_member */* returns the class defining this member */
);
```

See **os_member::defining_class()** on page 154.

os_member_function_create

```
extern os_member_function* os_member_function_create(
    char* /*name*/,
    os_function_type* /*function_type*/
);
```

See **os_member_function::create()** on page 159.

os_member_function_get_call_linkage

```
extern os_mop_call_linkage
os_member_function_get_call_linkage(
    os_member_function* /*function*/
);
```

See **os_member_function::get_call_linkage()** on page 159.

os_member _function_get_function_kind

```
extern os_function_kind_os_mop_function_kind
os_member_function_get_function_kind(
    os_member_function* /*function*/
);
```

See **os_member_function::get_function_kind()** on page 159.

os_member_function_get_name

```
extern char* os_member_function_get_name(
    os_member_function *
);
```

See **os_member_function::get_name()** on page 160.

os_member_function_get_source_position

```
extern void os_member_function_get_source_position(
    os_member_function* /*function*/,
    os_char_p* /*file*/,
    os_unsigned_int32* /*line*/
);
```

See **os_member_function::get_source_position()** on page 160.

os_member_function_get_type

```
extern os_function_type* os_member_function_get_type(
    os_member_function *
);
```

See **os_member_function::get_type()** on page 160.

os_member_function_is_const

```
extern os_boolean os_member_function_is_const(
    os_member_function *
);
```

See **os_member_function::is_const()** on page 160.

os_member_function_is_inline

```
extern os_boolean os_member_function_is_inline(
    os_member_function* /*function*/
);
```

See **os_member_function::is_inline()** on page 160.

os_member_function_is_overloaded

```
extern os_boolean os_member_function_is_overloaded(
    os_member_function* /*function*/
);
```

See **os_member_function::is_overloaded()** on page 161.

os_member_function_is_pure_virtual

```
extern os_boolean os_member_function_is_pure_virtual(
    os_member_function *
);
```

See **os_member_function::is_pure_virtual()** on page 161.

os_member_function_is_static

```
extern os_boolean os_member_function_is_static(
    os_member_function *
);
```

See **os_member_function::is_static()** on page 161.

os_member_function_is_virtual

```
extern os_boolean os_member_function_is_virtual(
    os_member_function *
);
```

See **os_member_function::is_virtual()** on page 161.

os_member_function_is_volatile

```
extern os_boolean os_member_function_is_volatile(
    os_member_function *
);
```

See **os_member_function::is_volatile()** on page 161.

os_member_function_set_call_linkage

```
extern void os_member_function_set_call_linkage(
    os_member_function* /*function*/,
    os_mop_call_linkage /*call_linkage*/
);
```

See **os_member_function::set_call_linkage()** on page 161.

os_member_function_set_function_kind

```
extern void os_member_function_set_function_kind(
    os_member_function* /*function*/,
    os_function_kind_os_mop_function_kind /*function_kind*/
);
```

See **os_member_function::set_function_kind()**.

os_member_function_set_is_const

```
extern void os_member_function_set_is_const(
    os_member_function* /*function*/,
    os_boolean /*is_const*/
);
```

See **os_member_function::set_is_const()** on page 161.

os_member_function_set_is_inline

```
extern void os_member_function_set_is_inline(
    os_member_function* /*function*/,
    os_boolean /*is_inline*/
);
```

See **os_member_function::set_is_inline()** on page 162.

os_member_function_set_is_overloaded

```
extern void os_member_function_set_is_overloaded(
    os_member_function* /*function*/,
    os_boolean /*is_overloaded*/
);
```

See **os_member_function::set_is_overloaded()** on page 162.

os_member_function_set_is_pure_virtual

```
extern void os_member_function_set_is_pure_virtual(
    os_member_function* /*function*/,
    os_boolean /*is_pure_virtual*/
);
```

See **os_member_function::set_is_pure_virtual()** on page 162.

os_member_function_set_is_static

```
extern void os_member_function_set_is_static(
    os_member_function* /*function*/,
    os_boolean /*is_static*/
);
```

See **os_member_function::set_is_static()** on page 162.

os_member_function_set_is_virtual

```
extern void os_member_function_set_is_virtual(
    os_member_function* /*function*/,
    os_boolean /*is_virtual*/
);
```

See **os_member_function::set_is_virtual()** on page 162.

os_member_function_set_is_volatile

```
extern void os_member_function_set_is_volatile(
    os_member_function* /*function*/,
    os_boolean /*is_volatile*/
);
```

See **os_member_function::set_is_volatile()** on page 162.

os_member_function_set_name

```
extern void os_member_function_set_name(
    os_member_function* /*function*/,
    char* /*name*/
);
```

See **os_member_function::set_name()** on page 162.

os_member_function_set_source_position

```
extern void os_member_function_set_source_position(
    os_member_function* /*function*/,
    /*const*/ char* /*file*/,
    /*const*/ os_unsigned_int32 /*line*/
);
```

See **os_member_function::set_source_position()** on page 162.

os_member_function_set_type

```
extern void os_member_function_set_type(
    os_member_function* /*function*/,
    os_function_type* /*function_type*/
);
```

See **os_member_function::set_type()** on page 163.

os_member_get_access

```
extern os_mop_member_access os_member_get_access(
    os_member *
);
```

See **os_member::get_access()** on page 154.

os_member_get_kind

```
extern os_mop_member_kind os_member_get_kind(
    os_member *
);
```

See **os_member::get_kind()** on page 155.

os_member_get_storage_class

```
extern os_mop_storage_class os_member_get_storage_class(
    os_member_variable* /*member_variable*/
);
```

See **os_member::get_storage_class()**.

os_member_is_unspecified

```
extern os_boolean os_member_is_unspecified(
    os_member* /*member*/
);
```

See **os_member::is_unspecified()** on page 155.

os_member_set_access

```
extern void os_member_set_access(
    os_member* /*member*/,
    os_unsigned_int32 /*access*/
    ) /*throw (err_mop)*/;
```

See **os_member::set_access()** on page 158.

os_member_type_cast_to_os_member

```
extern os_member* os_member_type_cast_to_os_member(
    os_member_type *
);
```

Cast to supertype.

os_member_type_create

```
extern os_member_type* os_member_type_create(
    os_type* /*nested_type*/
);
```

See **os_member_type::create()** on page 165.

os_member_type_get_type

```
extern os_type* os_member_type_get_type(
    os_member_type */* Return the type of this member */
);
```

See **os_member_type::get_type()** on page 165.

os_member_type_set_type

```
extern void os_member_type_set_type(
    os_member_type* /*member_type*/,
    os_type* /*nested_type*/
);
```

See **os_member_type::set_type()**.

os_member_variable_create

```
extern os_member_variable* os_member_variable_create(
    char* /*name*/,
    os_type* /*type*/
);
```

See **os_member_variable::create()** on page 166.

os_member_variable_get_name

```
extern char* os_member_variable_get_name(
    os_member_variable */* the name associated with this member */
);
```

See **os_member_variable::get_name()** on page 166.

os_member_variable_get_offset

```
extern os_unsigned_int32 os_member_variable_get_offset(
    os_member_variable */* the offset of this data member */
                /* relative to the defining class */
);
```

See **os_member_variable::get_offset()** on page 167.

os_member_variable_get_size

```
extern os_unsigned_int32 os_member_variable_get_size(
    os_member_variable */* the size in bytes of this */
                /* os_member_variable*/
);
```

See **os_member_variable::get_size()** on page 167.

os_member_variable_get_source_position

```
extern void os_member_variable_get_source_position(
    os_member_variable* /*member_variable*/,
    os_char_p* /*file*/,
    os_unsigned_int32* /*line*/
);
```

See **os_member_variable::get_source_position()** on page 167.

os_member_variable_get_type

```
extern os_type* os_member_variable_get_type(
    os_member_variable */* the type associated with this member */
);
```

See **os_member_variable::get_type()** on page 166.

os_member_variable_is_field

```
extern os_boolean os_member_variable_is_field(
    os_member_variable */* return true iff this member is */
                /* a bit field*/
);
```

See **os_member_variable::is_field()** on page 167.

os_member_variable_is_persistent

```
extern os_boolean os_member_variable_is_persistent(
    os_member_variable */* return true iff this is a */
                /* persistent data member*/
);
```

See **os_member_variable::is_persistent()** on page 168.

os_member_variable_is_static

```
extern os_boolean os_member_variable_is_static(
    os_member_variable */* return true iff this is a static */
                /* data member*/
);
```

See **os_member_variable::is_static()** on page 167.

os_member_variable_set_name

```
extern void os_member_variable_set_name(
    os_member_variable* /*member_variable*/,
    char* /*name*/
);
```

See **os_member_variable::set_name()** on page 168.

os_member_variable_set_offset

```
extern void os_member_variable_set_offset(
    os_member_variable* /*member_variable*/,
    os_unsigned_int32 /*offset*/
);
```

See **os_member_variable::set_offset()** on page 169.

os_member_variable_set_source_position

```
extern void os_member_variable_set_source_position(
    os_member_variable* /*member_variable*/,
    /*const*/ char* /*file*/,
    /*const*/ os_unsigned_int32 /*line*/
);
```

See **os_member_variable::set_source_position()** on page 169.

os_member_variable_set_storage_class

```
extern void os_member_variable_set_storage_class(
    os_member_variable* /*member_variable*/,
    os_unsigned_int32 /*storage_class*/
);
```

See **os_member_variable::set_storage_class()** on page 169.

os_member_variable_set_type

```
extern void os_member_variable_set_type(
    os_member_variable* /*member_variable*/,
    os_type* /*type*/
);
```

See **os_member_variable::set_type()** on page 169.

os_mop_copy_classes

```
extern void os_mop_copy_classes(
    /*const*/ os_schema* /*schema*/,
    os_collection* /*os_const_classes* classes*/
    ) /*throw (err_mop)*/;
```

See **os_mop::copy_classes()** on page 171.

os_mop_find_type

```
extern os_type* os_mop_find_type(
    /*const*/ char* /*name*/
);
```

See **os_mop::find_type()** on page 172.

os_mop_get_transient_schema

extern os_schema* os_mop_get_transient_schema();

See **os_mop::get_transient_schema()** on page 172.

os_mop_initialize

extern void os_mop_initialize();

See **os_mop::initialize()** on page 172.

os_mop_initialize_object_metadata

```
extern void os_mop_initialize_object_metadata(
    void */*object*/,
    const char */*type_name*/
);
```

See **os_mop::initialize_object_metadata()** on page 172.

os_named_indir_type_cast_to_is_indir_type

```
extern os_indirect_type*
os_named_indir_type_cast_to_os_indir_type(
    os_named_indirect_type *
);
```

Cast to supertype.

os_named_indir_type_cast_to_os_type

```
extern os_type* os_named_indir_type_cast_to_os_type(
    os_named_indirect_type *
);
```

Cast to supertype.

os_named_indir_type_get_target_type

```
extern os_type* os_named_indir_type_get_target_type(
    os_named_indirect_type *
);
```

See **os_indirect_type::get_target_type()** on page 144.

os_named_indirect_type_create

```
extern os_named_indirect_type* os_named_indirect_type_create(
    os_type* /*target_type*/,
    char* /*name*/
);
```

See **os_named_indirect_type::create()** on page 174.

os_named_indirect_type_get_name

```
extern char* os_named_indirect_type_get_name(
    os_named_indirect_type *
);
```

See **os_named_indirect_type::get_name()** on page 174.

os_named_indirect_type_get_source_position

```
extern void os_named_indirect_type_get_source_position(
    os_named_indirect_type*,
    os_char_p* /*file*/,
    os_unsigned_int32* /*line*/
);
```

See **os_named_indirect_type::get_source_position()** on page 174.

os_named_indirect_type_set_name

```
extern void os_named_indirect_type_set_name(
    os_named_indirect_type* /*indirect_type*/,
    char* /*name*/
);
```

See **os_named_indirect_type::set_name()** on page 174.

os_named_indirect_type_set_source_position

```
extern void os_named_indirect_type_set_source_position(
    os_named_indirect_type* /*indirect_type*/,
    /*const*/ char* /*file*/,
    /*const*/ os_unsigned_int32 /*line*/
);
```

See **os_named_indirect_type::set_source_position()** on page 174.

os_pointer_literal_create

```
extern os_pointer_literal* os_pointer_literal_create(
    char* /*name*/,
    os_pointer_type* /*ptr_type*/
);
```

See **os_pointer_literal::create()** on page 192.

os_pointer_literal_get_name

```
extern char* os_pointer_literal_get_name(
    os_pointer_literal *
);
```

See **os_pointer_literal::get_name()** on page 192.

os_pointer_literal_get_type

```
extern os_pointer_type* os_pointer_literal_get_type(
    os_pointer_literal *
);
```

See **os_pointer_literal::get_type()** on page 192.

os_pointer_literal_set_name

```
extern void os_pointer_literal_set_name(
    os_pointer_literal* /*literal*/,
    char* /*name*/
);
```

See **os_pointer_literal::set name()** on page 192.

os_pointer_literal_set_type

```
extern void os_pointer_literal_set_type(
    os_pointer_literal* /*literal*/,
    os_pointer_type* /*ptr_type*/
);
```

See **os_pointer_literal::set_type()** on page 192.

os_pointer_to_member_type_create

```
extern os_pointer_to_member_type*
os_pointer_to_member_type_create(
    os_type* /*target_type*/,
    os_class_type* /*class_type*/
);
```

See **os_pointer_to_member-type::create()** on page 193.

os_pointer_to_member_type_get_target_class

```
extern os_class_type*
os_pointer_to_member_type_get_target_class(
    os_pointer_to_member_type *
);
```

See **os_pointer_to_member_type::get_target_class()** on page 193.

os_pointer_to_member_type_set_target_class

```
extern void os_pointer_to_member_type_set_target_class(
    os_pointer_to_member_type* /*pointer_type*/,
    os_class_type* /*class_type*/
);
```

See **os_pointer_to_member_type::set_target_class()** on page 193.

os_pointer_type_cast_to_os_reference_type

```
extern os_reference_type*
os_pointer_type_cast_to_os_reference_type(
    os_pointer_type *
);
```

See **os_pointer_type** conversion operators.

os_pointer_type_cast_to_os_type

```
extern os_type* os_pointer_type_cast_to_os_type(
    os_pointer_type *
);
```

Cast to supertype.

os_pointer_type_create

```
extern os_pointer_type* os_pointer_type_create(
    os_type* /*target_type*/
);
```

See **os_pointer_type::create()** on page 194.

os_pointer_type_get_target_type

```
extern os_type* os_pointer_type_get_target_type(
    os_pointer_type *
);
```

See **os_pointer_type::get_target_type()** on page 194.

os_pointer_type_set_target_type

```
extern void os_pointer_type_set_target_type(
    os_pointer_type* /*pointer_type*/,
    os_type* /*target_type*/
);
```

See **os_pointer_type::set_target_type()** on page 194.

os_pragma_create_pragma

```
extern os_pragma* os_pragma_create_pragma(
    const char*
);
```

See **os_pragma::create()** on page 195.

os_pragma_get_string

```
extern char* os_pragma_get_string(
    os_pragma*
);
```

See **os_pragma::get_string()** on page 195.

os_pragma_is_recognized

```
extern os_boolean os_pragma_is_recognized(
    os_pragma*
);
```

See **os_pragma::is_recognized()** on page 195.

os_ptr_to_mem_type_cast_to_os_ptr_type

```
extern os_pointer_type* os_ptr_to_mem_type_cast_to_os_ptr_
type(
    os_pointer_to_member_type *
);
```

Cast to supertype.

os_ptr_to_mem_type_cast_to_os_type

```
extern os_type* os_ptr_to_mem_type_cast_to_os_type(
    os_pointer_to_member_type *
);
```

Cast to supertype.

os_ptr_type_cast_to_os_ptr_to_mem_type

```
extern os_pointer_to_member_type*
os_ptr_type_cast_to_os_ptr_to_mem_type(
    os_pointer_type *
);
```

See **os_pointer_to_member_type** conversion operators.

os_real_type_cast_to_os_type

```
extern os_type* os_real_type_cast_to_os_type(
    os_real_type *
);
```

Cast to supertype.

os_real_type_create

```
extern os_real_type* os_real_type_create(
    const char*
);
```

See **os_real_type::create()** on page 202.

os_real_type_create_double

```
extern os_real_type* os_real_type_create_double();
```

See **os_real_type::create_double()** on page 202.

os_real_type_create_float

```
extern os_real_type* os_real_type_create_float();
```

See **os_real_type::create_float()** on page 202.

os_real_type_create_long_double

```
extern os_real_type* os_real_type_create_long_double();
```

See **os_real_type::create_long_double()** on page 202.

os_ref_type_cast_to_os_ptr_type

```
extern os_pointer_type* os_ref_type_cast_to_os_ptr_type(
    os_reference_type *
);
```

Cast to supertype.

os_ref_type_cast_to_os_type

```
extern os_type* os_ref_type_cast_to_os_type(
    os_reference_type *
);
```

Cast to supertype.

os_reference_type_create

```
extern os_reference_type* os_reference_type_create(
    os_type* /*target_type*/
);
```

See **os_reference_type::create()** on page 266.

os_rel_mem_var_cast_to_os_mem_variable

```
extern os_member_variable*
```

```
os_rel_mem_var_cast_to_os_mem_variable(
    os_relationship_member_variable *
);
```

Cast to **os_member_variable**.

os_rel_mem_var_cast_to_os_member

```
extern os_member* os_rel_mem_var_cast_to_os_member(
    os_relationship_member_variable *
);
```

Cast to **os_member**.

os_rel_mem_var_get_related_class

```
extern os_class_type* os_rel_mem_var_get_related_class(
    os_relationship_member_variable *
);
```

See **os_relationship_member_variable::get_related_class()** on page 267.

os_rel_mem_var_get_related_member

```
extern os_relationship_member_variable*
os_rel_mem_var_get_related_member(
    os_relationship_member_variable *
);
```

See **os_relationship_member_variable::get_related_member()** on page 267.

os_schema_cast_to_os_app_schema

```
extern os_app_schema* os_schema_cast_to_os_app_schema(
    os_schema *
);
```

See **os_schema::operator os_app_schema&()** on page 271 and **os_schema::operator const os_app_schema&()** on page 270.

os_schema_cast_to_os_comp_schema

```
extern os_comp_schema* os_schema_cast_to_os_comp_schema(
    os_schema *
);
```

See **os_schema::operator os_comp_schema&()** and **os_schema::operator const os_comp_schema&()** on page 271.

os_schema_cast_to_os_database_schema

```
extern os_database_schema*os_schema_cast_to_os_database_
schema(
    os_schema *
);
```

See **os_schema::operator os_database_schema&()** and **os_schema::operator const os_database_schema&()** on page 271.

os_schema_find_type

```
extern os_type* os_schema_find_type(
    os_schema *,
    const char */* name of thing whose type we want */
);
```

See **os_schema::find_type()** on page 270.

os_schema_get_classes

```
extern os_collection* os_schema_get_classes(
    os_schema *
);
```

See **os_schema::get_classes()** on page 270.

os_schema_get_kind

```
extern os_mop_schema_kind os_schema_get_kind(
    os_schema* /*schema*/
);
```

See **os_schema::get_kind()** on page 270.

os_schema_install_options_create

```
extern void os_schema_install_options* os_schema_install_options_create();
```

Allocate an **os_schema_install_options** object. Caller must free the object with **os_schema_install_options_destroy**. See **os_schema_install_options::os_schema_install_options()** on page 294.

os_schema_install_options_destroy

```
extern void os_schema_install_options_destroy os_schema_
install_options_destroy
(os_schema_install_options*);
```

os_schema_install_options_get_copy_member_functions

```
extern void os_boolean
```

os_schema_install_options_get_copy_member_functions (os_ schema_install_options*);

See **os_schema_install_options::get_copy_member_functions ()** on page 294.

os_schema_install_options_set_copy_member_functions

extern void os_schema_install_options_set_copy_member_ functions (os_schema_install_options*, os_boolean);

See **os_schema_install_options::set_copy_member_functions ()** on page 294.

os_template_actual_arg_get_kind

```
extern os_mop_template_actual_arg_kind
os_template_actual_arg_get_kind(
    os_template_actual_arg* /*arg*/
);
```

See **os_template_actual_arg::get_kind()** on page 323.

os_template_formal_arg_get_kind

```
extern os_mop_template_formal_arg_kind
os_template_formal_arg_get_kind(
    os_template_formal_arg* /*arg*/
);
```

See **os_template_formal_arg::get_kind()** on page 325.

os_template_formal_arg_get_name

```
extern char* os_template_formal_arg_get_name(
    os_template_formal_arg* /*arg*/
);
```

See **os_template_formal_arg::get_name()** on page 325.

os_template_formal_arg_set_name

```
extern void os_template_formal_arg_set_name(
    os_template_formal_arg* /*arg*/,
    char* /*name*/
);
```

See **os_template_formal_arg::set_name()** on page 326.

os_template_formal_create

```
extern os_template_type_formal* os_template_formal_create(
    char* /*name*/
);
```

See **os_template_type_formal::create()** on page 329.

os_template_get_args

```
extern os_collection* os_template_get_args(
    os_template *
);
```

See **os_template::get_args()** on page 321.

os_template_get_kind

```
extern os_mop_template_kind os_template_get_kind(
    os_template */* Return the kind of this template */
);
```

See **os_template::get_kind()** on page 321.

os_template_instantiation_create

```
extern os_template_instantiation*
os_template_instantiation_create(
    os_template* /*a_template*/,
    os_collection* /*template_actual_args*/
);
```

See **os_template_instantiation::create()** on page 327.

os_template_instantiation_set_args

```
extern void os_template_instantiation_set_args(
    os_template_instantiation* /*instantiation*/,
    os_collection* /*template_actual_args*/
);
```

See **os_template_instantiation::set_args()** on page 327.

os_template_instantiation_set_template

```
extern void os_template_instantiation_set_template(
    os_template_instantiation* /*instantiation*/,
    os_template* /*a_template*/
);
```

See **os_template_instantiation::set_template()** on page 328.

os_template_is_unspecified

```
extern os_boolean os_template_is_unspecified(
    os_template* /*a_template*/
);
```

See **os_template::is_unspecified()** on page 321.

os_template_set_args

```
extern void os_template_set_args(
    os_template* /*a_template*/,
    os_collection* /*template_formal_args*/
);
```

See **os_template::set_args()** on page 322.

os_template_value_formal_create

```
extern os_template_value_formal*os_template_value_formal_
create(
    char* /*name*/,
    os_type* /*type*/
);
```

See **os_template_value_formal::create()** on page 330.

os_template_value_formal_get_type

```
extern os_type* os_template_value_formal_get_type(
    os_template_value_formal* /*formal*/
);
```

See **os_template_value_formal::get_type()** on page 330.

os_template_value_formal_set_type

```
extern void os_template_value_formal_set_type(
    os_template_value_formal* /*formal*/,
    os_type* /*type*/
);
```

See **os_template_value_formal::set_type()** on page 330.

os_tmplt_cast_to_os_type_tmplt

```
extern os_type_template* os_tmplt_cast_to_os_type_tmplt(
    os_template *
);
```

See **os_template::operator os_type_template&()** and **os_template::operator const os_type_template&()** on page 322.

os_type_cast_to_os_anonymous_indirect_type

```
extern os_anonymous_indirect_type*
    os_type_cast_to_os_anonymous_indirect_type(
    os_type *
```

);

See **os_type::operator os_anonymous_indirect_type&()** on page 348 and **os_type::operator const os_anonymous_indirect_type&()** on page 346.

os_type_cast_to_os_array_type

```
extern os_array_type* os_type_cast_to_os_array_type(
    os_type *
);
```

See **os_type::operator os_array_type&()** on page 348 and **os_type::operator const os_array_type&()** on page 346.

os_type_cast_to_os_class_type

```
extern os_class_type* os_type_cast_to_os_class_type(
    os_type *
);
```

See **os_type::operator os_class_type&()** on page 348 and **os_type::operator const os_class_type&()** on page 346.

os_type_cast_to_os_enum_type

```
extern os_enum_type* os_type_cast_to_os_enum_type(
    os_type *
);
```

See **os_type::operator os_enum_type&()** on page 348 and **os_type::operator const os_enum_type&()** on page 346.

os_type_cast_to_os_function_type

```
extern os_function_type* os_type_cast_to_os_function_type(
    os_type *
);
```

See **os_type::operator os_function_type&()** on page 348 and **os_type::operator const os_function_type&()** on page 346.

os_type_cast_to_os_instantiated_class_type

```
extern os_instantiated_class_type*
    os_type_cast_to_os_instantiated_class_type(
    os_type *
);
```

See **os_type::operator os_instantiated_class_type&()** on page 348 and **os_type::operator const os_instantiated_class_type&()** on page 346.

os_type_cast_to_os_integral_type

```
extern os_integral_type* os_type_cast_to_os_integral_type(
    os_type *
);
```

See **os_type::operator os_integral_type&()** on page 348 and **os_type::operator const os_integral_type&()** on page 347.

os_type_cast_to_os_named_indirect_type

```
extern os_named_indirect_type*
os_type_cast_to_os_named_indirect_type(
    os_type *
);
```

See **os_type::operator os_named_indirect_type&()** on page 349 and **os_type::operator const os_named_indirect_type&()** on page 347.

os_type_cast_to_os_pointer_to_member_type

```
extern os_pointer_to_member_type*os_pointer_to_member_type*
    os_type_cast_to_os_pointer_to_member_type(
    os_type *
);
```

See **os_type::operator os_pointer_to_member_type&()** on page 349 and **os_type::operator const os_pointer_to_member_type&()** on page 347.

os_type_cast_to_os_pointer_type

```
extern os_pointer_type* os_type_cast_to_os_pointer_type(
    os_type *
);
```

See **os_type::operator os_pointer_type&()** on page 349 and **os_type::operator const os_pointer_type&()** on page 347.

os_type_cast_to_os_real_type

```
extern os_real_type* os_type_cast_to_os_real_type(
    os_type *
);
```

See **os_type::operator os_real_type&()** on page 349 and **os_type::operator const os_real_type&()** on page 347.

os_type_cast_to_os_reference_type

```
extern os_reference_type* os_type_cast_to_os_reference_type(
    os_type *
);
```

See **os_type::operator os_reference_type&()** on page 349 and **os_ type::operator const os_reference_type&()** on page 347.

os_type_cast_to_os_type_type

```
extern os_type_type* os_type_cast_to_os_type_type(
    os_type *
);
```

See **os_type::operator os_type_type&()** on page 349 and **os_ type::operator const os_type_type&()** on page 347.

os_type_cast_to_os_void_type

```
extern os_void_type* os_type_cast_to_os_void_type(
    os_type *
);
```

See **os_type::operator os_void_type&()** on page 349 and **os_ type::operator const os_void_type&()** on page 348.

os_type_get_alignment

```
extern os_unsigned_int32 os_type_get_alignment(
    os_type */* return the alignment associated with this type */
);
```

See **os_type::get_alignment()** on page 343.

os_type_get_enclosing_class

```
extern os_class_type* os_type_get_enclosing_class(
    os_type* /*type*/
);
```

See **os_type::get_enclosing_class()** on page 343.

os_type_get_kind

```
extern os_mop_type_kind os_type_get_kind(
    os_type *
);
```

See **os_type::get_kind()** on page 343.

os_type_get_kind_string

```
extern char* os_type_get_kind_string(
    os_mop_type_kind/* the os_mop_type_kind enumerator */
);
```

See **os_type::get_kind_string()** on page 343.

os_type_get_size

```
extern os_unsigned_int32 os_type_get_size(
    os_type */* return the size (in bytes) of this type */
);
```

See **os_type::get_size()** on page 344.

os_type_get_string

```
extern char* os_type_get_string(
    os_type *
);
```

See **os_type::get_string()** on page 344.

os_type_is_const

```
extern os_boolean os_type_is_const(
    os_type *
);
```

See **os_type::is_const()** on page 344.

os_type_is_integral_type

```
extern os_boolean os_type_is_integral_type(
    os_type *
);
```

See **os_type::is_integral_type()** on page 345.

os_type_is_real_type

```
extern os_boolean os_type_is_real_type(
    os_type *p
);
```

See **os_type::is_real_type()** on page 345.

os_type_is_unspecified

```
extern os_boolean os_type_is_unspecified(
    os_type* /*type*/
);
```

See **os_type::is_unspecified()** on page 345.

os_type_is_volatile

```
os_boolean os_type_is_volatile
extern os_boolean os_type_is_volatile(
    os_type *
);
```

See **os_type::is_volatile()** on page 345.

os_type_set_alignment

```
extern void os_type_set_alignment(
    os_type* /*type*/,
    os_unsigned_int32 /*alignment*/
    ) /*throw (err_mop)*/;
```

See **os_type::set_alignment()** on page 350.

os_type_set_size

```
extern void os_type_set_size(
    os_type* /*type*/,
    os_unsigned_int32 /*size*/
    ) /*throw (err_mop)*/;
```

See **os_type::set_size()** on page 350.

os_type_strip_indirect_types

```
extern os_type* os_type_strip_indirect_types(
    os_type *
);
```

See **os_type::strip_indirect_types()** on page 350.

os_type_template_actual_arg

```
extern os_type_template_actual_arg*
os_type_template_actual_arg_create(
    os_type* /*type*/
);
```

See **os_type_template_actual_arg::create()** on page 355.

os_type_template_actual_arg_get_type

```
extern os_type* os_type_template_actual_arg_get_type(
    os_type_template_actual_arg* /*arg*/
);
```

See **os_type_template_actual_arg::get_type()** on page 355.

os_type_template_actual_arg_set_type

```
extern void os_type_template_actual_arg_set_type(
    os_type_template_actual_arg* /*arg*/,
    os_type* /*type*/
);
```

See **os_type_template_actual_arg::set_type()** on page 355.

os_type_template_create

```
extern os_type_template* os_type_template_create(
    os_type* /*type*/,
    os_collection* /*template_formal_args*/
);
```

See **os_type_template::create()** on page 354.

os_type_template_get_type

```
extern os_type* os_type_template_get_type(
    os_type_template *
);
```

See **os_type_template::get_type()** on page 354.

os_type_template_set_type

```
extern void os_type_template_set_type(
    os_type_template* /*type_template*/,
    os_type* /*type*/
);
```

See **os_type_template::set_type()** on page 354.

os_type_tmplt_cast_to_os_tmplt

```
extern os_template* os_type_tmplt_cast_to_os_tmplt(
    os_type_template *
);
```

Cast to supertype.

os_type_type_at

```
extern os_type* os_type_type_at(
    const void */* address of item whose type is to be identified */
);
```

See **os_type::type_at()** on page 352.

os_type_type_cast_to_os_type

```
extern os_type* os_type_type_cast_to_os_type(
    os_type_type *
);
```

Casts an **os_type_type** to an **os_type()**.

os_type_type_containing

```
extern os_type* os_type_type_containing(
    const void *,/* address of object (input) */
    os_void_const_p*,/* address (returned) of outermost object */
                /* containing inputted object*/
    os_unsigned_int32 */* address of the number of elements in */
                /* array if inputted object is an array */
);
```

See **os_type::type_containing()** on page 352.

os_type_type_create

```
extern os_type_type* os_type_type_create();
```

See **os_type_type::create()**.

os_void_type_cast_to_os_type

```
extern os_type* os_void_type_cast_to_os_type(
    os_void_type *
);
```

Cast to supertype.

os_void_type_create

```
extern os_void_type* os_void_type_create();
```

See **os_void_type::create()** on page 363.

os_notification Functions

The class definitions of **os_notification** and **os_subscription** are not visible in C. This presents particular difficulty for array operations, where **sizeof()** information is not available. To work around this problem, **sizeof()** functions are provided so that C callers can manipulate arrays of notifications and subscriptions.

To delete ObjectStore references use **objectstore_delete()**.

The text below lists each of the C API functions and its C++ equivalent.

os_notification_new

os_notification * os_notification_new(void);

Creates a new uninitialized notification. See **os_notification::os_ notification()** on page 176.

os_notification_free

void os_notification_free(os_notification *);

Deletes a notification. This is equivalent to **operator delete**.

os_notification_new_array

os_notification * os_notification_new_array(os_int32);

Creates a new array of uninitialized notifications. See **os_ notification::os_notification()** on page 176.

os_notification_free_array

void os_notification_free_array(os_notification*);

Deletes an array of notifications. It is equivalent to **operator delete []**.

os_notification_sizeof

os_int32 os_notification_sizeof(void);

Returns the size of an **os_notification** object, for use in array manipulation in C. It is equivalent to **sizeof(os_notification)**.

os_notification_assign

void os_notification_assign(

```
    os_notification*,
    os_reference*,
    os_int32,
    char *
);
```

Note that this takes a pointer to an **os_reference**, not a reference. In C, this means that the reference must be allocated on the heap before calling this function. See **os_notification::assign()** on page 176.

os_notification_get_database

os_database * os_notification_get_database(os_notification*);

See **os_notification::get_database()** on page 176.

os_notification_get_reference

os_reference * os_notification_get_reference(os_notification*);

Unlike the C++ version, this allocates a reference on the transient heap that must later be deleted using **objectstore_delete()**. See **os_notification::get_reference()** on page 177.

os_notification_get_kind

os_int32 os_notification_get_kind(os_notification*);

See **os_notification::get_kind()** on page 177.

os_notification_get_string

char * os_notification_get_string(os_notification*);

See **os_notification::get_string()** on page 177.

os_notification_subscribe

void os_notification_subscribe(os_subscription *, os_int32);

See **os_notification::subscribe()** on page 182.

os_notification_unsubscribe

void os_notification_unsubscribe(os_subscription *, os_int32);

See **os_notification::unsubscribe()** on page 183.

os_notification_notify_immediate

void os_notification_notify_immediate(

```
    os_int32,
    os_notification *,
    os_int32 *
);
```

See **os_notification::notify_immediate()** on page 178.

os_notification_notify_on_commit

```
void os_notification_notify_on_commit(os_notification *);
```

Note that in the C interface, you must first create an **os_ notification** object to pass to **notify_on_commit**. See **os_ notification::notify_on_commit()** on page 179.

os_notification_set_queue_size

```
void os_notification_set_queue_size(os_unsigned_int32);
```

See **os_notification::set_queue_size()** on page 182.

os_notification_queue_status

```
void os_notification_queue_status(
        os_unsigned_int32 *queue_size,
        os_unsigned_int32 *count_pending_notifications,
        os_unsigned_int32 *count_queue_overflows);
```

See **os_notification::queue_status()** on page 180.

os_notification_receive

```
os_boolean os_notification_receive(
    os_notification **notification,
    os_int32 timeout
);
```

See **os_notification::receive()** on page 181.

os_notification__get_fd

```
os_int32 os_notification__get_fd(void);
```

See **os_notification::_get_fd();** on page 177.

os_subscription_new

```
os_subscription * os_subscription_new(void);
```

Creates a new uninitialized **os_subscription**. See **os_ subscription::assign()** on page 319.

os_subscription_free

void os_subscription_free(os_subscription *);

Deletes an **os_subscription**. It is equivalent to **operator delete**.

os_subscription_new_array

os_subscription * os_subscription_new_array(os_int32);

Creates a new array of uninitialized **os_subscription**s. See **os_ subscription::assign()** on page 319.

os_subscription_free_array

void os_subscription_free_array(os_subscription*);

Deletes an array of **os_subscriptions**. It is equivalent to **operator delete []**.

os_subscription_sizeof

os_int32 os_subscription_sizeof(void);

Returns the size of an **os_subscription** object, for use in array manipulation in C. It is equivalent to **sizeof(os_subscription)**.

os_subscription_assign_database

void os_subscription_assign_database(os_subscription *, os_ database *);

See **os_subscription::assign()** on page 319.

os_subscription_assign_segment

void os_subscription_assign_segment(os_subscription *, os_ segment *);

See **os_subscription::assign()** on page 319.

os_subscription_assign_object_cluster

void os_subscription_assign_object_cluster(os_subscription *, os_ object_cluster *);

See **os_subscription::assign()** on page 319.

os_subscription_assign_reference

void os_subscription_assign_reference(os_subscription *, os_ reference *, os_int32);

Note that this takes a pointer to an **os_reference**, not a reference. In C, this means that the reference must be allocated on the heap before calling this function. See **os_subscription::assign()** on page 319.

os_subscription_get_database

os_database * os_subscription_get_database(os_subscription *);

See **os_subscription::get_database()** on page 319.

os_object_cluster Functions

The C library interface contains functions analogous to those of the class **os_object_cluster** in the ObjectStore Class Library.

os_object_cluster_destroy

```
extern void os_object_cluster_destroy(
        os_object_cluster* cluster,
        os_forced_destroy option
);
```

See **os_object_cluster::destroy()** on page 186.

os_object_cluster_free

```
extern void os_object_cluster_free(
        os_object_cluster* cluster
);
```

See **os_object_cluster::destroy()** on page 186.

os_object_cluster_get_info

```
extern void os_object_cluster_get_info(
        os_object_cluster* cluster,
        os_int32 *cluster_size,
        os_int32 *free,
        os_int32 *contig_free
);
```

See **os_object_cluster::get_info()** on page 187.

os_object_cluster_is_empty

```
extern os_boolean os_object_cluster_is_empty(
        os_object_cluster* cluster
);
```

See **os_object_cluster::is_empty()** on page 187.

os_object_cluster_of

```
extern os_object_cluster* os_object_cluster_of(
        os_object_cluster* cluster,
        void const* obj
);
```

See **os_object_cluster::of()** on page 187.

os_object_cluster_segment_of

```
extern os_segment* os_object_cluster_segment_of(
        os_object_cluster* cluster
);
```

See **os_object_cluster::segment_of()** on page 187.

os_object_cursor Functions

The C library interface contains functions analogous to those of the class **os_object_cursor** in the ObjectStore Class Library.

os_object_cursor_create

```
extern os_object_cursor* os_object_cursor_create(
    const os_segment *seg
);
```

See **os_object_cursor::os_object_cursor()** on page 190.

os_object_cursor_current

```
extern os_boolean os_object_cursor_current(
    os_object_cursor *cursor,
    void **pointer,
    const os_type **type,
    os_int32 *count
);
```

See **os_object_cursor::current()** on page 189.

os_object_cursor_delete

```
extern void os_object_cursor_delete(
    os_object_cursor *cursor
);
```

See **os_object_cursor::~os_object_cursor()** on page 190.

os_object_cursor_first

```
extern void os_object_cursor_first(
    os_object_cursor *cursor
);
```

See **os_object_cursor::first()** on page 189.

os_object_cursor_more

```
extern os_boolean os_object_cursor_more(
    os_object_cursor *cursor
);
```

See **os_object_cursor::more()** on page 190.

os_object_cursor_next

```
extern void os_object_cursor_next(
    os_object_cursor *cursor
);
```

See **os_object_cursor::next()** on page 190.

os_object_cursor_set

```
extern void os_object_cursor_set(
    os_object_cursor *cursor,
    void *pointer
);
```

See **os_object_cursor::set()** on page 190.

os_pvar Functions

The C library interface contains functions analogous to those of the class **os_pvar** in the ObjectStore Class Library.

os_pvar_define

```
extern void os_pvar_define(
    database *,
    void **,
    char *,
    os_typespec *,
    os_pvar_init_function
);
```

See **os_pvar::os_pvar()** on page 197.

os_pvar_init_int

```
extern void * os_pvar_init_int(
    database *
);
```

See **os_pvar::init_int()** on page 198.

os_pvar_init_long

```
extern void * os_pvar_init_long(
    database *
);
```

See **os_pvar::init_long()** on page 198.

os_pvar_init_pointer

```
extern void * os_pvar_init_pointer(
    database *
);
```

See **os_pvar::init_pointer()** on page 198.

os_pvar_undefine

```
extern void os_pvar_undefine(
        void **
);
```

See **os_pvar::undefine()**.

os_rawfs_entry Functions

The C library interface contains functions analogous to those of the class **os_rawfs_entry** in the ObjectStore Class Library.

os_rawfs_entry_delete

```
void os_rawfs_entry_delete(
        os_rawfs_entry *stat_entry,
        os_boolean is_array);
```

Deletes an **os_rawfs_entry** struct and frees storage. See **os_rawfs_entry::~os_rawfs_entry()** on page 201.

os_rawfs_entry_get_creation_time

```
os_unixtime_t os_rawfs_entry_get_creation_time(
        const os_rawfs_entry* entry
        );
```

See **os_rawfs_entry::get_creation_time()** on page 199.

os_rawfs_entry_get_group_name

```
const char* os_rawfs_entry_get_group_name(
        const os_rawfs_entry* entry
        );
```

See **os_rawfs_entry::get_group_name()** on page 199.

os_rawfs_entry_get_link_host

```
const char* os_rawfs_entry_get_link_host(
        const os_rawfs_entry* entry
        );
```

See **os_rawfs_entry::get_link_host()** on page 199.

os_rawfs_entry_get_link_path

```
const char* os_rawfs_entry_get_link_path(
        const os_rawfs_entry* entry
        );
```

See **os_rawfs_entry::get_link_path()** on page 199.

os_rawfs_entry_get_n_sectors

```
extern os_unsigned_int32 os_rawfs_entry_get_n_sectors(
        const os_rawfs_entry* entry
        );
```

See **os_rawfs_entry::get_n_sectors()** on page 199.

os_rawfs_entry_get_name

```
const char* os_rawfs_entry_get_name(
        const os_rawfs_entry* entry
        );
```

See **os_rawfs_entry::get_name()** on page 199.

os_rawfs_entry_get_permission

```
extern os_unsigned_int32 os_rawfs_entry_get_permission(
        const os_rawfs_entry* entry
        );
```

See **os_rawfs_entry::get_permission()** on page 200.

os_rawfs_entry_get_server_host

```
const char* os_rawfs_entry_get_server_host(
        const os_rawfs_entry* entry
        );
```

See **os_rawfs_entry::get_server_host()** on page 200.

os_rawfs_entry_get_type

```
extern os_int32 os_rawfs_entry_get_type(
        const os_rawfs_entry* entry
        );
```

See **os_rawfs_entry::get_type()** on page 200.

os_rawfs_entry_get_user_name

```
const char* os_rawfs_entry_get_user_name(
        const os_rawfs_entry* entry
        );
```

See **os_rawfs_entry::get_user_name()** on page 200.

os_rawfs_entry_is_db

```
extern os_boolean os_rawfs_entry_is_db(
        const os_rawfs_entry* entry
        );
```

See **os_rawfs_entry::is_db()** on page 200.

os_rawfs_entry_is_dir

```
extern os_boolean os_rawfs_entry_is_dir(
        const os_rawfs_entry* entry
        );
```

See **os_rawfs_entry::is_dir()** on page 200.

os_rawfs_entry_is_link

```
extern os_boolean os_rawfs_entry_is_link(
        const os_rawfs_entry* entry
        );
```

See **os_rawfs_entry::is_link()** on page 200.

os_segment Functions

ObjectStore databases are divided into *segments*. Each segment can be used as a unit of transfer to and from persistent storage. Every database is created with one *initial* segment. Additional segments are created by the user, if desired.

os_segment_allow_external_pointers

```
extern void os_segment_allow_external_pointers(
        os_segment *
);
```

See **os_segment::allow_external_pointers()** on page 295.

os_segment_count_objects

```
extern void os_segment_count_objects(
        os_segment *,
        os_int32 *, /* length */
        objectstore_object_count **
);
```

See **os_segment::count_objects()**.

os_segment_create_object_cluster

```
extern os_object_cluster* os_segment_create_object_cluster(
        os_segment *seg, os_unsigned_int32 size
);
```

See **os_segment::create_object_cluster()** on page 296.

os_segment_database_of

```
extern os_database * os_segment_database_of(
        os_segment *
);
```

See **os_segment::database_of()** on page 296.

os_segment_destroy

```
extern void os_segment_destroy(
        os_segment *
);
```

See **os_segment::destroy()** on page 296.

os_segment_external_pointer_status

```
extern void os_segment_external_pointer_status(
        os_segment *,
        os_boolean *,
        os_boolean *
);
```

See **os_segment::external_pointer_status()** on page 297.

os_segment_get_access_control

```
extern os_segment_access* os_segment_get_access_control(
    os_segment *segment
);
```

See **os_segment::get_access_control()** on page 297.

os_segment_get_all_object_clusters

```
extern void os_segment_get_all_object_clusters(
        os_segment *seg, os_int32 max_clusters,
        os_object_cluster** cluster_array,
        os_int32* n_clusters
);
```

See **os_segment::get_all_object_clusters()** on page 297.

os_segment_get_application_info

```
extern void * os_segment_get_application_info(
        os_segment *
);
```

See **os_segment::get_application_info()** on page 297.

os_segment_get_check_illegal_pointers

```
extern os_boolean os_segment_get_check_illegal_pointers(
        os_segment *
);
```

See **os_segment::get_check_illegal_pointers()** on page 298.

os_segment_get_comment

```
extern char * os_segment_get_comment(
        os_segment *seg
);
```

See **os_segment::get_comment()** on page 298.

os_segment_get_database_references

```
extern void os_segment_get_database_references(
        os_segment *,
        os_int32 *,
        os_database_reference ***
);
```

See **os_segment::get_database_references()** on page 298.

os_segment_get_fetch_policy

```
extern void os_segment_get_fetch_policy(
        os_segment *,
        os_fetch_policy *,
        os_int32 *
);
```

See **os_segment::get_fetch_policy()** on page 298.

```
extern objectstore_lock_option objectstore_lock_option
os_segment_get_lock_whole_segment(
        os_segment *
);
```

See **os_segment::get_lock_whole_segment()** on page 298.

os_segment_get_n_object_clusters

```
extern os_int32 os_segment_get_n_object_clusters(
        os_segment *seg
);
```

See **os_segment::get_n_object_clusters()** on page 299.

os_segment_get_null_illegal_pointers

```
extern os_boolean os_segment_get_null_illegal_pointers(
        os_segment *
);
```

See **os_segment::get_null_illegal_pointers()** on page 299.

os_segment_get_number

```
extern os_unsigned_int32 os_segment_get_number(
        os_segment *
);
```

See **os_segment::get_number()** on page 299.

os_segment_get_readlock_timeout

```
extern os_int32 os_segment_get_readlock_timeout(
        os_segment *seg
);
```

See **os_segment::get_readlock_timeout()** on page 299.

os_segment_get_size

```
extern os_unsigned_int32 os_segment_get_size(
        os_segment *
);
```

See **os_segment::get_size()**.

os_segment_get_transient_segment

```
os_segment * os_segment_get_transient_segment();
```

See **os_segment::get_transient_segment()** on page 299.

os_segment_get_writelock_timeout

```
extern os_int32 os_segment_get_writelock_timeout(
        os_segment *seg
);
```

See **os_segment::get_writelock_timeout()** on page 299.

os_segment_is_empty

```
extern os_boolean os_segment_is_empty(
        os_segment *
);
```

See **os_segment::is_empty()** on page 300.

os_segment_lock_into_cache

```
extern void os_segment_lock_into_cache(
        os_segment *
);
```

See **os_segment::lock_into_cache()** on page 300.

os_segment_of

```
extern os_segment * os_segment_of(
        void *
);
```

See **os_segment::of()** on page 300.

os_segment_return_memory

```
extern os_unsigned_int32 os_segment_return_memory(
          os_segment *,
          os_boolean
);
```

See **os_segment::return_memory()** on page 300.

os_segment_set_access_control

```
extern void os_segment_set_access_control(
   os_segment *segment,
   const os_segment_access *new_access
);
```

See **os_segment::set_access_control()** on page 300.

os_segment_set_application_info

```
extern void os_segment_set_application_info(
          os_segment *,
          void *
);
```

See **os_segment::set_application_info()** on page 301.

os_segment_set_check_illegal_pointers

```
extern void os_segment_set_check_illegal_pointers(
          os_segment *,
          os_boolean
);
```

See **os_segment::set_check_illegal_pointers()** on page 301.

os_segment_set_comment

```
extern void os_segment_set_comment(
          os_segment *seg,
          char *new_comment
);
```

See **os_segment::set_comment()** on page 301.

os_segment_set_fetch_policy

```
extern void os_segment_set_fetch_policy(
          os_segment *,
          os_fetch_policy,
          os_int32
);
```

See **os_segment::set_fetch_policy()** on page 302.

os_segment_set_lock_whole_segment

```
extern void os_segment_set_lock_whole_segment(
        os_segment *,
        objectstore_lock_option
);
```

See **os_segment::set_lock_whole_segment()** on page 303.

os_segment_set_null_illegal_pointers

```
extern void os_segment_set_null_illegal_pointers(
        os_segment *,
        os_boolean
);
```

See **os_segment::set_null_illegal_pointers()** on page 304.

os_segment_set_readlock_timeout

```
extern void os_segment_set_readlock_timeout(
        os_segment *seg,
        os_int32 milliseconds
);
```

See **os_segment::set_readlock_timeout()** on page 304.

os_segment_set_size

```
extern os_unsigned_int32 os_segment_set_size(
        os_segment *,
        os_unsigned_int32
);
```

See **os_segment::set_size()** on page 305.

os_segment_set_writelock_timeout

```
extern void os_segment_set_writelock_timeout(
        os_segment *seg,
        os_int32 milliseconds
);
```

See **os_segment::set_writelock_timeout()** on page 305.

os_segment_size

```
extern os_unsigned_int32 os_segment_size(
        os_segment *
);
```

See **os_segment::size()** on page 305.

os_segment_unlock_from_cache

```
extern void os_segment_unlock_from_cache(
        os_segment *
);
```

See **os_segment::unlock_from_cache()** on page 305.

os_segment_access Functions

The C library interface contains functions analogous to those of the class **os_segment_access** in the ObjectStore Class Library.

os_segment_access_delete

```
extern void os_segment_access_delete(
    os_segment_access *control,
    os_boolean is_array
);
```

See **os_segment_access::~os_segment_access()** on page 310.

os_segment_access_get_default

```
extern os_int32 os_segment_access_get_default(
    os_segment_access *control
);
```

See **os_segment_access::get_default()** on page 307.

os_segment_access_get_primary_group

```
extern os_int32 os_segment_access_get_primary_group(
    os_segment_access *control,
    const char** group
);
```

See **os_segment_access::get_primary_group()** on page 308.

os_segment_access_set_default

```
extern void os_segment_access_set_default(
    os_segment_access *control,
    os_int32 type
);
```

See **os_segment_access::set_default()** on page 309.

os_segment_access_set_primary_group

```
extern void os_segment_access_set_primary_group(
    os_segment_access *control,
    const char* group,
    os_int32 type
);
```

See **os_segment_access::set_primary_group()** on page 309.

os_server Functions

The C library interface contains functions analogous to those of the class **os_server** in the ObjectStore Class Library.

os_server_connection_is_broken

```
extern os_boolean os_server_connection_is_broken(
    os_server *
);
```

See **os_server::connection_is_broken()** on page 312.

os_server_disconnect

```
extern void os_server_disconnect(
    os_server *
);
```

See **os_server::disconnect()** on page 312.

os_server_get_databases

```
extern void os_server_get_databases(
    os_server *,
    os_int32,/* max db's to return */
    os_database_p*, /* where to return them */
    os_int32 */* how many returned */
);
```

See **os_server::get_databases()** on page 312.

os_server_get_host_name

```
extern char * os_server_get_host_name(
    os_server *
);
```

See **os_server::get_host_name()** on page 312.

os_server_get_n_databases

```
extern os_int32 os_server_get_n_databases(
    os_server *
);
```

See **os_server::get_n_databases()** on page 313.

os_server_reconnect

```
extern void os_server_reconnect(
    os_server *
);
```

See **os_server::reconnect()** on page 313.

os_transaction Functions

The C library interface contains functions analogous to those of the class **os_transaction** in the ObjectStore Class Library.

os_transaction_abort

```
extern void os_transaction_abort(
        os_transaction *
);
```

See **os_transaction::abort()** on page 331.

os_transaction_abort_top_level

```
extern void os_transaction_abort_top_level();
```

See **os_transaction::abort_top_level()** on page 332.

os_transaction_begin

```
extern os_transaction * os_transaction_begin(
        transaction_type
);
```

See **os_transaction::begin()** on page 332.

os_transaction_begin_named

```
extern os_transaction * os_transaction_begin_named(
char* name, os_transaction_type
);
```

See **os_transaction::begin()** on page 332.

os_transaction_commit

```
extern void os_transaction_commit(
        os_transaction *
);
```

See **os_transaction::commit()** on page 335.

os_transaction_get_abort_handle

```
extern tix_exception * os_transaction_get_abort_handle(
        os_transaction *
);
```

See **os_transaction::get_abort_handle()**.

os_transaction_get_current

```
extern os_transaction * os_transaction_get_current();
```

See **os_transaction::get_current()** on page 335.

os_transaction_get_exception_status

```
extern tix_exception * os_transaction_get_exception_status(
        os_transaction *
);
```

See **os_transaction::get_exception_status()**.

os_transaction_get_max_retries

```
os_int32 os_transaction_get_max_retries();
```

See **os_transaction::get_max_retries()** on page 336.

os_transaction_get_name

```
extern char* os_transaction_get_name(
os_transaction*
);
```

See **os_transaction::get_name()** on page 336.

os_transaction_get_parent

```
extern os_transaction * os_transaction_get_parent(
        os_transaction *
);
```

See **os_transaction::get_parent()** on page 336.

os_transaction_get_type

```
extern os_transaction_type os_transaction_get_type(
os_transaction *
);
```

See **os_transaction::get_type()** on page 336.

os_transaction_is_aborted

```
extern os_boolean os_transaction_is_aborted(
os_transaction *
);
```

See **os_transaction::is_aborted()** on page 336.

os_transaction_is_committed

extern os_boolean os_transaction_is_committed(
os_transaction *
);

See **os_transaction::is_committed()** on page 336.

os_transaction_set_max_retries

extern void os_transaction_set_max_retries(
os_int32
);

See **os_transaction::set_max_retries()** on page 338.

os_transaction_set_name

extern void os_transaction_set_name(
os_transaction*, const char*
);

See **os_transaction::set_name()** on page 338.

os_transaction_top_level

extern os_boolean os_transaction_top_level(
os_transaction *
);

See **os_transaction::top_level()** on page 338.

os_typespec Functions

os_typespec_equal

```
extern long os_typespec_equal(
    os_typespec *,
    os_typespec *
);
```

Returns nonzero if the specified typespecs designate the same type; **0** otherwise.

os_typespec_get_char

```
extern os_typespec * os_typespec_get_char();
```

See **os_typespec::get_char()** on page 358.

os_typespec_get_double

```
extern os_typespec * os_typespec_get_double();
```

See **os_typespec::get_double()** on page 358.

os_typespec_get_float

```
extern os_typespec * os_typespec_get_float();
```

See **os_typespec::get_float()** on page 358.

os_typespec_get_int

```
extern os_typespec * os_typespec_get_int();
```

See **os_typespec::get_int()** on page 359.

os_typespec_get_long

```
extern os_typespec * os_typespec_get_long();
```

See **os_typespec::get_long()** on page 359.

os_typespec_get_long_double

```
extern os_typespec * os_typespec_get_long_double();
```

See **os_typespec::get_long_double()** on page 359.

os_typespec_get_name

```
extern char * os_typespec_get_name(
    os_typespec *
```

);

Returns the name of the type designated by the specified typespec.

os_typespec_get_pointer

extern os_typespec * os_typespec_get_pointer();

See **os_typespec::get_pointer()** on page 359.

os_typespec_get_short

extern os_typespec * os_typespec_get_short();

See **os_typespec::get_short()** on page 360.

os_typespec_get_signed_char

os_typespec_get_signed_char
extern os_typespec * os_typespec_get_signed_char();

See **os_typespec::get_signed_char()** on page 360.

os_typespec_get_signed_int

extern os_typespec * os_typespec_get_signed_int();

See **os_typespec::get_signed_int()** on page 360.

os_typespec_get_signed_long

extern os_typespec * os_typespec_get_signed_long();

See **os_typespec::get_signed_long()** on page 360.

os_typespec_get_signed_short

extern os_typespec * os_typespec_get_signed_short();

See **os_typespec::get_signed_short()** on page 361.

os_typespec_get_unsigned_char

extern os_typespec * os_typespec_get_unsigned_char();

See **os_typespec::get_unsigned_char()** on page 361.

os_typespec_get_unsigned_int

extern os_typespec * os_typespec_get_unsigned_int();

See **os_typespec::get_unsigned_int()** on page 361.

os_typespec_get_unsigned_long

extern os_typespec * os_typespec_get_unsigned_long();

See **os_typespec::get_unsigned_long()** on page 361.

os_typespec_get_unsigned_short

extern os_typespec * os_typespec_get_unsigned_short();

See **os_typespec::get_unsigned_short()** on page 361.

os_typespec_name_is

```
extern os_boolean os_typespec_name_is(
          os_typespec *,
          char *
);
```

See **os_typespec::name_is()**.

Reference Functions

The C library interface contains functions analogous to those of the classes **os_reference**, **os_reference_local**, **os_reference_ protected**, **os_reference_protected_local**, **os_reference_this_DB**, and **os_reference_transient** in the ObjectStore Class Library.

os_reference_EQ

```
extern int os_reference_EQ(
    os_reference* ref1,
    os_reference* ref2
);
```

See **os_reference::operator ==()** on page 212.

os_reference_GE

```
extern int os_reference_GE(
    os_reference* ref1,
    os_reference* ref2
);
```

See **os_reference::operator >=()** on page 213.

os_reference_GT

```
extern int os_reference_GT(
    os_reference* ref1,
    os_reference* ref2
);
```

See **os_reference::operator >()** on page 213.

os_reference_LE

```
extern int os_reference_LE(
    os_reference* ref1,
    os_reference* ref2
);
```

See **os_reference::operator <=()** on page 213.

os_reference_LT

```
extern int os_reference_LT(
    os_reference* ref1,
    os_reference* ref2
);
```

See **os_reference::operator <()** on page 213.

os_reference_NE

```
extern int os_reference_NE(
    os_reference* ref1,
    os_reference* ref2
);
```

See **os_reference::operator !=()** on page 212.

os_reference_NOT

```
extern int os_reference_NOT(
    os_reference* ref1
);
```

Returns nonzero if the reference is null; **0** otherwise.

os_reference_dump

```
extern char* os_reference_dump(
    os_reference* ref
);
```

See **os_reference::dump()** on page 210.

os_reference_dump2

```
extern char* os_reference_dump2(
    os_reference* ref,
    char *db_str
);
```

See **os_reference::dump()** on page 210.

os_reference_get_database_key

```
extern char* os_reference_get_database_key(
    os_reference* ref,
    const char* dump_str
);
```

See **os_reference::get_database_key()** on page 211.

os_reference_load2

```
extern void os_reference_load2(
    os_reference* ref,
    const char *handle,
    const os_database *db
);
```

See **os_reference::load()** on page 211.

os_reference_new

```
extern os_reference* os_reference_new(
    void* ptr, segment* seg
);
```

See **os_reference::os_reference()** on page 213.

os_reference_new_from_string

```
extern os_reference* os_reference_new_from_string(
    char* string, segment* seg
);
```

See **os_reference::os_reference()** on page 213.

os_reference_resolve

```
extern void* os_reference_resolve(
    os_reference* ref
);
```

See **os_reference::resolve()** on page 214.

os_reference_local_EQ

```
extern int os_reference_local_EQ(
    os_reference_local* ref1,
    os_reference_local* ref2
);
```

See **os_reference_local::operator ==()** on page 222.

os_reference_local_GE

```
extern int os_reference_local_GE(
    os_reference_local* ref1,
    os_reference_local* ref2
);
```

See **os_reference_local::operator >=()** on page 222.

os_reference_local_GT

```
extern int os_reference_local_GT(
    os_reference_local* ref1,
    os_reference_local* ref2
);
```

See **os_reference_local::operator >()** on page 222.

os_reference_local_LE

```
extern int os_reference_local_LE(
    os_reference_local* ref1,
    os_reference_local* ref2
);
```

See **os_reference_local::operator <=()** on page 223.

os_reference_local_LT

```
extern int os_reference_local_LT(
    os_reference_local* ref1,
    os_reference_local* ref2
);
```

See **os_reference_local::operator <()** on page 222.

os_reference_local_NE

```
extern int os_reference_local_NE(
    os_reference_local* ref1,
    os_reference_local* ref2
);
```

See **os_reference_local::operator !=()** on page 222.

os_reference_local_NOT

```
extern int os_reference_local_NOT(
    os_reference_local* ref1
);
```

Returns nonzero if the reference is null; **0** otherwise.

os_reference_local_dump

```
extern char* os_reference_local_dump(
    os_reference_local* ref, char* database_name
);
```

See **os_reference_local::dump()** on page 221.

os_reference_local_get_database_key

```
extern char* os_reference_local_get_database_key(
    os_reference_local* ref,
    char* dump_str
);
```

See **os_reference_local::get_database_key()** on page 221.

os_reference_local_new

```
extern os_reference_local* os_reference_local_new(
    void* ptr, segment* seg
);
```

Creates a reference in the specified segment.

os_reference_local_new_from_string

```
extern os_reference_local* os_reference_local_new_from_string(
    char* string, segment* seg
);
```

See **os_reference_local::os_reference_local()** on page 223.

os_reference_local_resolve

```
extern void* os_reference_local_resolve(
    os_reference_local* ref, database* database
);
```

See **os_reference_local::resolve()** on page 223.

os_reference_protected_EQ

```
extern int os_reference_protected_EQ(
    os_reference_protected* ref1,
    os_reference_protected* ref2
);
```

See **os_reference_protected::operator ==()** on page 234.

os_reference_protected_GE

```
extern int os_reference_protected_GE(
    os_reference_protected* ref1,
    os_reference_protected* ref2
);
```

See **os_reference_protected::operator >=()** on page 235.

os_reference_protected_GT

```
extern int os_reference_protected_GT(
    os_reference_protected* ref1,
    os_reference_protected* ref2
);
```

See **os_reference_protected::operator >()** on page 235.

os_reference_protected_LE

```
extern int os_reference_protected_LE(
    os_reference_protected* ref1,
    os_reference_protected* ref2
);
```

See **os_reference_protected::operator <=()** on page 235.

os_reference_protected_LT

```
extern int os_reference_protected_LT(
    os_reference_protected* ref1,
    os_reference_protected* ref2
);
```

See **os_reference_protected::operator <()** on page 235.

os_reference_protected_NE

```
extern int os_reference_protected_NE(
    os_reference_protected* ref1,
    os_reference_protected* ref2
);
```

See **os_reference_protected::operator !=()** on page 235.

os_reference_protected_deleted

```
extern int os_reference_protected_deleted(
    os_reference_protected* ref
);
```

See **os_reference_protected::deleted()** on page 232.

os_reference_protected_dump

```
extern char* os_reference_protected_dump(
    os_reference_protected* ref
);
```

See **os_reference_protected::dump()** on page 232.

os_reference_protected_dump2

```
extern char* os_reference_protected_dump2(
    os_reference_protected* ref,
    const char *db_str
);
```

See **os_reference_protected::dump()** on page 232.

os_reference_protected_forget

```
extern void os_reference_protected_forget(
    os_reference_protected* ref
);
```

See **os_reference_protected::forget()** on page 233.

os_reference_protected_get_database_key

```
extern char* os_reference_protected_get_database_key(
    os_reference_protected* ref,
    const char* dump_str
);
```

See **os_reference_protected::get_database_key();** on page 233.

os_reference_protected_load2

```
extern char* os_reference_protected_load2(
    os_reference* ref,
    const char *handle,
    const os_database *db
);
```

See **os_reference_protected::load()** on page 234.

os_reference_protected_local_EQ

```
extern int os_reference_protected_local_EQ(
    os_reference_protected_local* ref1,
    os_reference_protected_local* ref2
);
```

See **os_reference_protected_local::operator ==()** on page 244.

os_reference_protected_local_GE

```
extern int os_reference_protected_local_GE(
    os_reference_protected_local* ref1,
    os_reference_protected_local* ref2
);
```

See **os_reference_protected_local::operator >=()** on page 245.

os_reference_protected_local_GT

```
extern int os_reference_protected_local_GT(
    os_reference_protected_local* ref1,
    os_reference_protected_local* ref2
);
```

See **os_reference_protected_local::operator >()** on page 245.

os_reference_protected_local_LE

```
extern int os_reference_protected_local_LE(
    os_reference_protected_local* ref1,
    os_reference_protected_local* ref2
);
```

See **os_reference_protected_local::operator <=()** on page 245.

os_reference_protected_local_LT

```
extern int os_reference_protected_local_LT(
    os_reference_protected_local* ref1,
    os_reference_protected_local* ref2
);
```

See **os_reference_protected_local::operator <()** on page 245.

os_reference_protected_local_NE

```
extern int os_reference_protected_local_NE(
    os_reference_protected_local* ref1,
    os_reference_protected_local* ref2
);
```

See **os_reference_protected_local::operator !=()** on page 244.

os_reference_protected_local_deleted

```
extern int os_reference_protected_local_deleted(
    os_reference_protected_local* ref,
        database* db
);
```

See **os_reference_protected_local::deleted()** on page 243.

os_reference_protected_local_dump

```
extern char* os_reference_protected_local_dump(
    os_reference_protected_local* ref,
    char* database_name
);
```

See **os_reference_protected_local::dump()** on page 243.

os_reference_protected_local_forget

```
extern void os_reference_protected_local_forget(
    os_reference_protected_local* ref,
        database* db
);
```

See **os_reference_protected_local::forget()** on page 243.

os_reference_protected_local_get_database_key

extern char* os_reference_protected_local_get_database_key(
    os_reference_protected_locak* ref,
    const char* dump_str
);

See **os_reference_protected_local::get_database_key()** on page 243.

extern os_reference_protected_local* os_reference_protected_ local*
os_reference_protected_local_new(
    void* ptr, segment* seg
);

Creates a reference in the specified segment.

extern os_reference_protected_local* os_reference_protected_ local*
os_reference_protected_local_new_from_string(
    char* string, segment* seg
);

See **os_reference_protected_local::os_reference_protected_local()** on page 246.

os_reference_protected_local_resolve

extern void* os_reference_protected_local_resolve(
    os_reference_protected_local* ref,
    database* database
);

See **os_reference_protected_local::resolve()** on page 246.

os_reference_protected_new

extern os_reference_protected* os_reference_protected_new(
void* ptr, segment* seg
);

Creates a reference in the specified segment.

os_reference_protected_new_from_string

extern os_reference_protected*
os_reference_protected_new_from_string(
    char* string, segment* seg
);

See **os_reference_protected::os_reference_protected()** on page 236.

os_reference_protected_resolve

```
extern void* os_reference_protected_resolve(
    os_reference_protected* ref
);
```

See **os_reference_protected::resolve()** on page 236.

os_reference_this_DB_EQ

```
extern int os_reference_this_DB_EQ(
    os_reference_this_DB* ref1,
    os_reference_this_DB* ref2
);
```

See **os_reference_this_DB::operator ==()** on page 254.

os_reference_this_DB_GE

```
extern int os_reference_this_DB_GE(
    os_reference_this_DB* ref1,
    os_reference_this_DB* ref2
);
```

See **os_reference_this_DB::operator >=()** on page 255.

os_reference_this_DB_GT

```
extern int os_reference_this_DB_GT(
    os_reference_this_DB* ref1,
    os_reference_this_DB* ref2
);
```

See **os_reference_this_DB::operator >()** on page 255.

os_reference_this_DB_LE

```
extern int os_reference_this_DB_LE(
    os_reference_this_DB* ref1,
    os_reference_this_DB* ref2
);
```

See **os_reference_this_DB::operator <=()** on page 255.

os_reference_this_DB_LT

```
extern int os_reference_this_DB_LT(
    os_reference_this_DB* ref1,
    os_reference_this_DB* ref2
);
```

See **os_reference_this_DB::operator <()** on page 255.

os_reference_this_DB_NE

```
extern int os_reference_this_DB_NE(
    os_reference_this_DB* ref1,
    os_reference_this_DB* ref2
);
```

See **os_reference_this_DB::operator !=()** on page 254.

os_reference_this_DB_NOT

```
extern int os_reference_this_DB_NOT(
    os_reference_this_DB* ref1
);
```

Returns nonzero if the specified reference is null; **0** otherwise.

os_reference_this_DB_dump

```
extern char* os_reference_this_DB_dump(
    os_reference_this_DB* ref
);
```

See **os_reference_this_DB::dump()** on page 253.

os_reference_this_DB_new

```
extern os_reference_this_DB* os_reference_this_DB_new(
    void* ptr, segment* seg
);
```

Creates a reference in the specified segment.

os_reference_this_DB_new_from_string

```
extern os_reference_this_DB*
os_reference_this_DB_new_from_string(
    char* string, segment* seg
);
```

See **os_reference_this_DB::os_reference_this_DB()** on page 255.

os_reference_this_DB_resolve

```
extern void* os_reference_this_DB_resolve(
    os_reference_this_DB* ref
);
```

See **os_reference_this_DB::resolve()** on page 256.

os_reference_this_DB_dump2

```
extern char* os_reference_this_DB_dump2(
    os_reference_this_DB* ref,
    const char* db_str
);
```

See **os_reference_this_DB::dump()** on page 253.

os_reference_transient_EQ

```
extern int os_reference_transient_EQ(
    os_reference_transient* ref1,
    os_reference_transient* ref2
);
```

See **os_reference_transient::operator ==()** on page 264.

os_reference_this_DB_get_database_key

```
extern char* os_reference_this_DB_get_database_key(
    os_reference_this_DB* ref,
    const char* dump_str
);
```

See **os_reference_this_DB::get_database_key()** on page 253.

os_reference_transient_GE

```
os_reference_transient_GE
extern int os_reference_transient_GE(
    os_reference_transient* ref1,
    os_reference_transient* ref2
);
```

See **os_reference_transient::operator >=()** on page 265.

os_reference_transient_GT

```
extern int os_reference_transient_GT(
    os_reference_transient* ref1,
    os_reference_transient* ref2
);
```

See **os_reference_transient::operator >()** on page 264.

os_reference_transient_LE

```
extern int os_reference_transient_LE(
    os_reference_transient* ref1,
    os_reference_transient* ref2
);
```

See **os_reference_transient::operator <=()** on page 265.

os_reference_transient_LT

```
extern int os_reference_transient_LT(
    os_reference_transient* ref1,
    os_reference_transient* ref2
);
```

See **os_reference_transient::operator <()** on page 264.

os_reference_transient_NE

```
extern int os_reference_transient_NE(
    os_reference_transient* ref1,
    os_reference_transient* ref2
);
```

See **os_reference_transient::operator !=()** on page 264.

os_reference_transient_NOT

```
extern int os_reference_transient_NOT(
    os_reference_transient* ref1
);
```

Returns nonzero if the specified reference is null; **0** otherwise.

os_reference_transient_dump

```
extern char* os_reference_transient_dump(
    os_reference_transient* ref
);
```

See **os_reference_transient::dump()** on page 262.

os_reference_transient_new

```
extern os_reference_transient* os_reference_transient_new(
    void* ptr, segment* seg
);
```

Creates a reference in the specified segment.

os_reference_transient_new_from_string

```
extern os_reference_transient*
os_reference_transient_new_from_string(
    char* string, segment* seg
);
```

See **os_reference_transient::os_reference_transient()** on page 265.

os_reference_transient_resolve

```
extern void* os_reference_transient_resolve(
    os_reference_transient* ref
);
```

See **os_reference_transient::resolve()** on page 265.

os_reference_transient_dump2

```
extern char* os_reference_transient_dump2(
    os_reference_transient* ref,
    const char* db_str
);
```

See **os_reference_transient::dump()** on page 262.

os_reference_transient_get_database_key

```
extern char* os_reference_transient_get_database_key(
    os_reference* ref,
    const char* dump_str);
```

See **os_reference_transient::get_database_key()** on page 263.

os_reference_transient_load2

```
extern char* os_reference_transient_load2(
    os_reference_transient* ref,
    const char *handle,
    const os_database *db
);
```

See **os_reference_transient::load()** on page 263.

Schema Evolution Functions

The following functions provide the C interface to ObjectStore schema evolution. See **os_schema_evolution** in the *ObjectStore Advanced C++ API User Guide*.

os_evolve_subtype_fun_binding_create

```
extern os_evolve_subtype_fun_binding* os_evolve_subtype_fun_
binding_create(
    char* type,
    os_evolve_subtype_fun fcn,
    char* fcn_name
);
```

See **os_evolve_subtype_fun_binding::os_evolve_subtype_fun_binding()** on page 138.

os_evolve_subtype_fun_binding_destroy

```
extern void os_evolve_subtype_fun_binding_destroy(
    os_evolve_subtype_fun_binding*
);
```

Deletes the specified **os_evolve_subtype_fun_binding**.

os_prim_typed_pointer_void_get_pointer

```
extern void* os_prim_typed_pointer_void_get_pointer(
    os_typed_pointer_void*/* this */
);
```

See **os_typed_pointer_void::get_pointer()**.

os_prim_typed_pointer_void_get_type

```
extern os_type* os_prim_typed_pointer_void_get_type(
    os_typed_pointer_void*/* this */
);
```

See **os_typed_pointer_void::get_type()** on page 357.

os_schema_evolution_augment_classes_to_be_recycled

```
extern void os_schema_evolution_augment_classes_to_be_
recycled(
    char* /* name of class to be recycled */
);
```

See **os_schema_evolution::augment_classes_to_be_recycled()** on page 276.

os_schema_evolution_augment_classes_to_be_recycled_multiple

```
extern void
os_schema_evolution_augment_classes_to_be_recycled_multiple(
    os_collection<char*>*/* names of classes to be recycled */
);
```

See **os_schema_evolution::augment_classes_to_be_recycled()** on page 276.

os_schema_evolution_augment_classes_to_be_removed

```
extern void os_schema_evolution_augment_classes_to_be_
removed(
    char*     /* name of class to be removed */
);
```

See **os_schema_evolution::augment_classes_to_be_removed()** on page 277.

os_schema_evolution_augment_classes_to_be_removed_multiple

```
extern void os_schema_evolution_augment_classes_to_be_
removed_multiple(
    os_collection<char*>*/* names of classes to be removed */
);
```

See **os_schema_evolution::augment_classes_to_be_removed()** on page 277.

os_schema_evolution_augment_nonversioned_transformers

```
extern void os_schema_evolution_augment_nonversioned_
transformers(
os_transformer_binding*
);
```

See **os_schema_evolution::augment_nonversioned_transformers()**.

os_schema_evolution_augment_nonversioned_transformers_multiple

```
extern void os_schema_evolution_augment_nonversioned_
transformers_multiple(
os_transformer_bindings*
);
```

See **os_schema_evolution::augment_nonversioned_transformers()**.

os_schema_evolution_augment_post_evol_transformers

```
extern void
os_schema_evolution_augment_post_evol_transformers(
```

os_transformer_binding*
);

See **os_schema_evolution::augment_post_evol_transformers()** on page 277.

os_schema_evolution_augment_post_evol_transformers_multiple

extern void os_schema_evolution_augment_post_evol_
transformers_multiple(
os_transformer_bindings*
);

See **os_schema_evolution::augment_post_evol_transformers()** on page 277.

os_schema_evolution_augment_pre_evol_transformers

extern void os_schema_evolution_augment_pre_evol_
transformers(
os_transformer_binding*
);

See **os_schema_evolution::augment_pre_evol_transformers()**.

os_schema_evolution_augment_pre_evol_transformers_multiple

extern void os_schema_evolution_augment_pre_evol_
transformers_multiple(
os_transformer_bindings*
);

See **os_schema_evolution::augment_pre_evol_transformers()**.

os_schema_evolution_augment_subtype_selectors

extern void os_schema_evolution_augment_subtype_selectors(
os_evolve_subtype_fun_binding*
);

See **os_schema_evolution::augment_subtype_selectors()** on page 278.

os_schema_evolution_augment_subtype_selectors_multiple

extern void os_schema_evolution_augment_subtype_selectors_
multiple(
os_evolve_subtype_fun_bindings*
);

See **os_schema_evolution::augment_subtype_selectors()** on page 278.

os_schema_evolution_evolve

```
extern void os_schema_evolution_evolve(
    char*,    /* name of work db */
    char*     /* name of db to be evolved */
);
```

See **os_schema_evolution::evolve()** on page 278.

os_schema_evolution_evolve_multiple

```
extern void os_schema_evolution_evolve_multiple(
    char*,    /* name of work db */
    os_charp_collection*/* names of dbs to be evolved */
);
```

See **os_schema_evolution::evolve()** on page 278.

os_schema_evolution_get_enclosing_object

```
extern os_typed_pointer_void
os_schema_evolution_get_enclosing_object(
    void*
);
```

See **os_schema_evolution::get_enclosing_object()**.

os_schema_evolution_get_evolved_address

```
extern os_typed_pointer_void
os_schema_evolution_get_evolved_address(
    void*     /* addr-1 */
);
```

See **os_schema_evolution::get_evolved_address()** on page 283.

os_schema_evolution_get_evolved_object

```
extern os_typed_pointer_void
os_schema_evolution_get_evolved_object(
    void*     /* object-1 */
);
```

See **os_schema_evolution::get_evolved_object()** on page 283.

os_schema_evolution_get_ignore_illegal_pointers

```
extern os_boolean
os_schema_evolution_get_ignore_illegal_pointers();
```

See **os_schema_evolution::get_ignore_illegal_pointers()** on page 283.

os_schema_evolution_get_obsolete_index_handler

```
extern os_obsolete_index_handler_fun
os_schema_evolution_get_obsolete_index_handler();
```

See **os_schema_evolution::get_obsolete_index_handler()**.

os_schema_evolution_get_obsolete_query_handler

```
extern os_obsolete_query_handler_fun
os_schema_evolution_get_obsolete_query_handler();
```

See **os_schema_evolution::get_obsolete_query_handler()**.

os_schema_evolution_get_path_to_member

```
extern os_path* os_schema_evolution_get_path_to_member(
          void*
);
```

See **os_schema_evolution::get_path_to_member()**.

os_schema_evolution_get_unevolved_address

```
extern os_typed_pointer_void
os_schema_evolution_get_unevolved_address(
   void*     /* addr-2 */
);
```

See **os_schema_evolution::get_unevolved_address()** on page 283.

os_schema_evolution_get_unevolved_object

```
extern os_typed_pointer_void
os_schema_evolution_get_unevolved_object(
   void*     /* object-2 */
);
```

See **os_schema_evolution::get_unevolved_object()** on page 284.

os_schema_evolution_get_work_database

```
extern os_database * os_schema_evolution_get_work_database();
```

See **os_schema_evolution::get_work_database()** on page 284.

os_schema_evolution_set_evolved_schema_db_name

```
extern void os_schema_evolution_set_evolved_schema_db_name(
   char*     /* evolved schema db name */
);
```

See **os_schema_evolution::set_evolved_schema_db_name()** on page 284.

os_schema_evolution_set_explanation_level

```
extern void os_schema_evolution_set_explanation_level(
            os_unsigned_int32
);
```

See **os_schema_evolution::set_explanation_level()**.

os_schema_evolution_set_ignore_illegal_pointers

```
extern void os_schema_evolution_set_ignore_illegal_pointers(
os_boolean
);
```

See **os_schema_evolution::set_ignore_illegal_pointers()** on page 284.

os_schema_evolution_set_illegal_pointer_handler_database_root

```
extern void
os_schema_evolution_set_illegal_pointer_handler_database_root(
    os_illegal_database_root_handler_fun
);
```

See **os_schema_evolution::set_illegal_pointer_handler()** on page 284.

os_schema_evolution_set_illegal_pointer_handler_pointer

```
extern void
os_schema_evolution_set_illegal_pointer_handler_pointer(
    os_illegal_pointer_handler_fun
);
```

See **os_schema_evolution::set_illegal_pointer_handler()** on page 284.

os_schema_evolution_set_illegal_pointer_handler_pointer_to_member

```
extern void
os_schema_evolution_set_illegal_pointer_handler_pointer
_to_member(
    os_illegal_pointer_to_member_handler_fun
);
```

See **os_schema_evolution::set_illegal_pointer_handler()** on page 284.

os_schema_evolution_set_illegal_pointer_handler_reference

```
extern void
os_schema_evolution_set_illegal_pointer_handler_reference(
    os_illegal_reference_handler_fun
);
```

See **os_schema_evolution::set_illegal_pointer_handler()** on page 284.

os_schema_evolution_set_illegal_pointer_handler_reference_local

```
extern void (
    os_illegal_reference_local_handler_fun
);
```

See **os_schema_evolution::set_illegal_pointer_handler()** on page 284.

os_schema_evolution_set_illegal_pointer_handler_reference_version

```
extern void (
    os_illegal_reference_version_handler_fun
);
```

See **os_schema_evolution::set_illegal_pointer_handler()** on page 284.

os_schema_evolution_set_local_references_are_db_relative

```
extern void os_schema_evolution_set_local_references_are_db_
relative(
    os_boolean
);
```

See **os_schema_evolution::set_local_references_are_db_relative()** on page 286.

os_schema_evolution_set_obsolete_index_handler

```
extern void os_schema_evolution_set_obsolete_index_handler(
    os_obsolete_index_handler_fun
);
```

See **os_schema_evolution::set_obsolete_index_handler()** on page 286.

os_schema_evolution_set_obsolete_query_handler

```
extern void os_schema_evolution_set_obsolete_query_handler(
    os_obsolete_query_handler_fun
);
```

See **os_schema_evolution::set_obsolete_query_handler()** on page 286.

os_schema_evolution_set_pre_evolution_setup_hook

```
extern void os_schema_evolution_set_pre_evolution_setup_hook(
os_hook_function_void
);
```

See **os_schema_evolution::set_task_list_file_name()** on page 286.

os_schema_evolution_set_recycle_all_classes

```
extern void os_schema_evolution_set_recycle_all_classes(
    os_boolean
);
```

See **os_schema_evolution::set_recycle_all_classes()**.

os_schema_evolution_set_resolve_ambiguous_void_pointers

```
extern void
os_schema_evolution_set_resolve_ambiguous_void_pointers(
os_boolean
);
```

See **os_schema_evolution::set_resolve_ambiguous_void_ pointers()**.

os_schema_evolution_set_task_list_file_name

```
extern void os_schema_evolution_set_task_list_file_name(
            char* file_name
);
```

See **os_schema_evolution::set_task_list_file_name()** on page 286.

os_schema_evolution_task_list

```
extern void os_schema_evolution_task_list(
    char*,    /* name of work db */
    char* /* db to be evolved */
);
```

See **os_schema_evolution::task_list()** on page 287.

os_schema_evolution_task_list_multiple

```
extern void os_schema_evolution_task_list_multiple(
    char*,    /* name of work db */
    os_charp_collection*/* names of dbs to be evolved */
);
```

See **os_schema_evolution::task_list()** on page 287.

os_transformer_binding_create

```
extern os_transformer_binding* os_transformer_binding_create(
    char* type,
    os_transformer fcn,
    char* fcn_name
);
```

See **os_transformer_binding::os_transformer_binding()** on page 341.

os_transformer_binding_destroy

```
extern void os_transformer_binding_destroy(
    os_transformer_binding*
);
```

Deletes the specified **os_transformer_binding**.

os_prim_typed_pointer_void_assign

```
extern void os_prim_typed_pointer_void_assign(
    os_typed_pointer_void*,/* this */
    void*,    /* thing to point to */
    os_type*/* type of the thing pointed to */
);
```

See **os_typed_pointer_void::os_typed_pointer_void()**.

TIX Functions

The C library interface contains functions analogous to those of the classes **tix_exception**, **tix_handler**, and **basic_undo**. See Appendix A, Exception Facility, on page 555.

os_delete_tix_exception

```
extern void os_delete_tix_exception(
tix_exception * /* excp */
);
```

Deletes an exception.

os_new_tix_exception

```
extern tix_exception * os_new_tix_exception(
char * /* message */, tix_exception * /* parent */
);
```

Creates an exception.

tix_exception_message

```
extern char * tix_exception_message(
    tix_exception * /* exception */
);
```

See **tix_exception::message()**.

tix_exception_parent

```
extern tix_exception * tix_exception_parent(
    tix_exception * /* exception */
);
```

See **tix_exception::parent()**.

tix_exception_signal

```
extern void tix_exception_signal(
    tix_exception * /* excp */,
    char */* format */,
    ...
);
```

See **tix_exception::signal()**.

tix_exception_signal_val

```
extern void tix_exception_signal_val(
    tix_exception * /* excp */,
    int      /* value */,
    char */* format */,
    ...
);
```

See **tix_exception::signal()**.

tix_exception_vsignal

```
extern void tix_exception_vsignal(
    tix_exception * /* excp */,
    int /* value */,
    char * /* format */,
    va_list /* args */
);
```

See **tix_exception::vsignal()**.

tix_handler_get_current

```
extern tix_handler *tix_handler_get_current();
```

See **tix_handler::get_current()** on page 561.

tix_handler_get_exception

```
extern tix_exception *tix_handler_get_exception();
```

See **tix_handler::get_exception()** on page 561.

tix_handler_get_report

```
extern char *tix_handler_get_report();
```

See **tix_handler::get_report()** on page 561.

tix_handler_get_value

```
extern int tix_handler_get_value();
```

See **tix_handler::get_value()** on page 562.

Appendix A
Exception Facility

ObjectStore supports the use of two kinds of exceptions:

- C++ exceptions
- TIX exceptions

C++ exceptions are a part of the C++ language supported by some C++ compilers. On any platform whose compiler supports exceptions, you can signal and catch C++ exceptions in ObjectStore applications.

TIX exceptions are part of the ObjectStore application programming interface. ObjectStore API functions sometimes signal predefined TIX exceptions when error conditions arise, and your programs can handle these exceptions with macros and member functions provided by ObjectStore. These predefined exceptions are listed in Appendix B, Predefined TIX Exceptions, on page 569.

In addition, you can

- Define new TIX exceptions
- Signal predefined or user-defined TIX exceptions
- Handle user-defined TIX exceptions

Macros

The macros described below provide a means of handling TIX exceptions, and defining and declaring variables of type **tix_exception** (see **tix_exception** on page 560). They also provide a way of specifying a handler for a TIX exception signaled by a given block of code.

For an example involving signaling and handling TIX exceptions, see Sample TIX Exception-Handling Routine on page 566.

TIX_HANDLE, TIX_EXCEPTION, and TIX_END_HANDLE

Three macros are used in conjunction to specify a handler for an exception signaled by a given code block:

```
TIX_HANDLE (identifier)
    statement-1...statement-n
TIX_EXCEPTION
    statement
TIX_END_HANDLE
```

identifier is the name of the TIX exception to be handled. This should not be **err_internal** — you cannot handle ObjectStore internal errors.

statement-1 ... statement-n are considered to be within their own block, as is *statement*. This code sequence indicates that if the exception *identifier* (or a *descendent* of it — see **tix_exception** on page 560) is signaled while control is dynamically within the block containing *statement-1*...*statement-n*, control is transferred to *statement*, unless another appropriate handler for the exception is found first (see **tix_exception** on page 560, **tix_exception::signal()**). If no exception is signaled, then, when the block containing *statement-1*...*statement-n* completes, control flows to the statement immediately following **TIX_END_HANDLE**.

Example

```
extern char db_name_buf[];
extern os_database *db;

void input_and_open()
{
    db = 0;

    while (!db) {
        get_db_name(db_name_buf) ;
        TIX_HANDLE (err_database_not_found)
```

```
            db = os_database::open(db_name_buf);
        TIX_EXCEPTION
            printf("Sorry, no such database. Try again.\n");
        TIX_END_HANDLE
    }
}
```

The call to the ObjectStore-supplied function **os_database::open()** can result in signaling err_database_not_found. In such a case, the call does not return, but rather control is transferred nonlocally to the **printf()** call, which constitutes the exception *handler*. Once the handler has been executed, control flows to the top of the loop.

These constructs can be nested, allowing handling of multiple exceptions. For example:

```
TIX_HANDLE (exc1)
    TIX_HANDLE (exc2)
        statements
    TIX_EXCEPTION
        handler statement for exc2
    TIX_END_HANDLE
TIX_EXCEPTION
    handler statement for exc1
TIX_END_HANDLE
```

TIX_HANDLE_IF

```
TIX_HANDLE_IF (flag, identifier)
    statement-1...statement-n
TIX_EXCEPTION
    statement
TIX_END_HANDLE
```

TIX_HANDLE_IF is similar to **TIX_HANDLE**, but it takes two arguments: *flag* and *identifier*. *flag* should be a C++ Boolean expression, as would be specified in **if** or **when** statements. If the expression evaluates to **true** (nonzero), the handler is established around the body; otherwise, no handler is established. *identifier* identifies the exception, just like the **TIX_HANDLE** argument.

DEFINE_EXCEPTION

```
#define DEFINE_EXCEPTION(name, message, pointer_to_parent)\
objectstore_exception name("name", message, pointer_to_parent)
```

To define a TIX exception, you call the macro **DEFINE_EXCEPTION()** as follows:

```
    DEFINE_EXCEPTION(name, message, pointer-to-parent)
```

name is the name (not in quotation marks) of the exception to be created.

message is a string to be printed when the exception is raised but there is no handler to catch it. Use quotation marks for this argument.

pointer-to-parent is the address of the existing exception you want to serve as the new exception's parent. Supply **0** if you do not want the exception to have a parent.

```
DEFINE_EXCEPTION(bad_char_input, "BAD CHARACTER", &bad_input)
```

DECLARE_EXCEPTION

```
#define DECLARE_EXCEPTION(name) \
extern objectstore_exception name
```

This macro is used to declare a variable of type **objectstore_ exception** (see **tix_exception** on page 560).

Classes

The exception facility provides five classes: **objectstore_exception, tix_exception**, **tix_handler**, **basic_undo**, and **tix_context**.

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.

objectstore_exception

When you use the macro **DEFINE_EXCEPTION** or **DECLARE_EXCEPTION** you define or declare a variable whose type is **objectstore_exception**. The class **tix_exception** is a base class of **objectstore_exception**, which inherits several useful functions from it.

objectstore_exception::objectstore_exception()

```
objectstore_exception(
    char *name,
    char const *message,
    tix_exception const *exc
);
```

Returns a newly constructed instance of **objectstore_exception**. **name** is the name of the exception. **message** is a message to be printed if the exception is raised but not handled. **exc** specifies the new exception’s parent.

objectstore_exception::signal()

```
void signal(char const *format ...);
```

Signals the TIX exception for which the function is called. The arguments work as with the function **printf()**. A format string is supplied, followed by any number of additional arguments. Control is transferred to the most recently established handler for the exception or one of its ancestors. If there is no such handler, the exception’s associated message is issued, together with the message specified by **format** and associated arguments. The process is then aborted or exited from, as determined by the environment variable **OS_DEF_EXCEPT_ACTION** (see *ObjectStore Management*).

```
void signal(os_int32 value, char const *format ...);
```

Signals the TIX exception for which the function is called. The first argument can be used to associate an error code with the signaling of the exception. The value can be retrieved with the function **tix_ handler::get_value()**. The subsequent arguments work as with the function **printf()**. A format string is supplied, followed by any number of additional arguments. Control is transferred to the most recently established handler for the exception or one of its ancestors. If there is no such handler, the exception's associated message is issued, together with the message specified by **format** and associated arguments, and the process is aborted.

objectstore_exception::vsignal()

void vsignal(char const *format, va_list args);

Signals the TIX exception for which the function is called. The arguments work as with the function **vprintf()**. A format string is supplied, followed by a **va_list**. Control is transferred to the most recently established handler for the exception or one of its ancestors. If there is no such handler, the exception's associated message is issued, together with the message specified by **format** and associated arguments. The process is then aborted or exited from, as determined by the environment variable **OS_DEF_ EXCEPT_ACTION** (see *ObjectStore Management*).

void vsignal(os_int32 value, char const *format, va_list args);

Signals the TIX exception for which the function is called. The first argument can be used to associate an error code with the signaling of the exception. The value can be retrieved with the function **tix_ handler::get_value()**. The subsequent arguments work as with the function **vprintf()**. A format string is supplied, followed by a **va_ list**. Control is transferred to the most recently established handler for the exception or one of its ancestors. If there is no such handler, the exception's associated message is issued, together with the message specified by **format** and associated arguments, and the process is aborted.

tix_exception

Every TIX exception is an instance of **tix_exception**, and represents a particular type of error condition or exceptional circumstance. Every exception has a *parent*, except the exception **err_objectstore** (see Appendix B, Predefined TIX Exceptions, on page 569). The parent of a given exception is an *ancestor* of that exception. The parent of any ancestor of a given exception is also an *ancestor* of that exception. If one exception is an ancestor of another, the other exception is said to be a *descendent* of the first.

tix_exception::ancestor_of()

os_int32 ancestor_of(tix_exception const *e);

Returns nonzero if **this** is an ancestor of **e**, and **0** otherwise. Any parent of **e** is an ancestor of **e**, and any parent of an ancestor of **e** is also an ancestor of **e**. In addition, every exception is considered to be its own ancestor, so if **this** is the same as **e**, the function returns nonzero.

tix_exception::set_unhandled_exception_hook()

```
static tix_unhandled_exception_hook_t
    set_unhandled_exception_hook(
    tix_unhandled_exception_hook_t new_hook
);
```

Registers a function that is called if a **tix_exception** is unhandled. I You must be sure not to access persistent memory while in err_ deadlock when this function is in use. If this happens it can lead to a recursive exception error message.

tix_handler

A **tix_handler** carries information about the particular exceptional circumstances leading to the signaling of the current exception.

tix_handler::get_current()

static tix_handler *get_current();

Returns the **tix_handler** for the current exception.

tix_handler::get_exception()

static tix_exception *get_exception();

Returns the current TIX exception.

tix_handler::get_report()

static char *get_report();

Returns the format string used to signal the current exception. The string is suitable for further processing or writing to a file, as opposed to being displayed on a screen. See also **tix_handler_get_ report0()**.

tix_handler::get_report0()

static char *get_report0();

Returns the format string used to signal the current exception. The string is suitable for displaying on a screen. See also **tix_handler_ get_report()**.

tix_handler::get_value()

static os_int32 get_value();

Returns the error code associated with the current exception.

basic_undo

In C++, the destructor for an automatic object is run whenever a stack frame containing it is unwound. When a C++ exception is signaled and control is transferred to a handler in an enclosing scope, the intervening stack frames are unwound. When a TIX exception is signaled and control is transferred to a handler in an enclosing scope, the intervening stack frames are unwound on some platforms, but not on others.

On the following platforms, the intervening stack frames *are* unwound:

- OS/2 VisualAge C++
- NT Visual C++
- Windows 95 VC++
- AXP Digital UNIX C++

On other platforms, you can derive a class from **basic_undo** to partially simulate the unwinding of the intervening stack frames. ObjectStore runs the destructors for instances of **basic_undo** contained in those stack frames.

The destructor for **basic_undo** is virtual. So the destructor for instances of classes derived from **basic_undo** is run when the unwinding is simulated. The destructors for these derived classes can perform any desired undo processing. You must create an instance of the derived class in the desired scope before an exception can be raised. Note that this object must be stack-allocated, not heap-allocated. The object's destructor will be run whenever the signaling of an exception transfers control out of or through its stack frame. (Of course, the destructor will also be run if the block is exited from normally.)

A class derived from **basic_undo** must have a virtual destructor if (and only if) further classes are derived from it.

basic_undo::basic_undo()

basic_undo();

Constructs a new instance of **basic_undo**.

basic_undo::exception

protected: tix_exception *exception;

Has as value a pointer to the TIX exception currently being signaled. If no exception is being signaled — that is, the block is being exited from normally — the value is **0**. Your destructor can access this to learn what is going on.

basic_undo::~basic_undo()

virtual ~basic_undo();

This destructor is invoked whenever a frame is unwound by **tix_exception::signal()**. Overloadings of it defined by derived classes can perform undo processing. This function is also invoked in the normal way upon block exit.

tix_context

The class **tix_context** makes error messages more informative by providing contextual information.

In the following example, if you get the error Connection attempt timed out, you are also told that the error occurred while you were trying to create that database.

```
{ tix_context tc1("While creating database %s\n", the_pathname);
server->create_database(the_pathname);
}
```

If any TIX exception is signaled during the execution of **server->create_database**, and the exception is not handled inside the **server->create_database**, the error message associated with the error will be extended to include the string **"While creating database /a/b/c"** at the end.

tix_context works as follows: when an exception is signaled, as the signaler moves up the stack searching for a handler, if it encounters a **tix_context**, it runs **sprintf** on the arguments, and adds this string to the end of the error message. If it does finally find a handler, the string returned by the **"report"** function member (and **"report0"**) will include the context messages of all **tix_context** objects that were on the stack between the point where the exception was signaled and the point where it was handled. If no handler is found, the message that is issued will include the context messages of all **tix_context** objects that were on the stack.

The **tix_context** constructor has one required argument, a **printf** control string, and up to four additional arguments, which must all be strings. The **printf** control string should have as many occurrences of **"%s"** as there are arguments. You can only use strings and **"%s"**, and no more than four.

C++ requires that you include a variable name for the **tix_context** object, such as **tc1** in the example above.

Note that the **sprintf** is done only if the exception is actually signaled, so that establishing a **tix_context** that never gets used incurs little overhead.

The **tix_context** constructor does *not* copy any of the strings it is given, so if there is any chance that those strings might get altered during the execution of the scope of the **tix_context**, you should probably make copies. It is a good idea to delete them manually when you are done with them.

The current limit on the length of the report string is 4096 characters.

os_lock_timeout_exception

Instances of this class contain information on the circumstances preventing acquisition of a lock within a specified timeout period. An exception of this type can be signaled by processes that have called the **set_readlock_timeout()** or **set_writelock_timeout()** member of **os_segment**, **os_database**, or **objectstore**. A pointer to an instance of this class can be returned by **objectstore::acquire_ lock()**.

os_lock_timeout_exception::get_application_names()

char **get_application_names();

Returns an array of strings naming the applications preventing lock acquisition.

os_lock_timeout_exception::get_fault_addr()

void *get_fault_addr();

Returns the address on which ObjectStore faulted, causing the database access leading to the attempted lock acquisition.

os_lock_timeout_exception::get_hostnames()

char **get_hostnames();

Returns an array of strings naming the host machines running the applications preventing lock acquisition.

os_lock_timeout_exception::get_lock_type()

os_lock_type get_lock_type();

Returns an enumerator (**os_read_lock** or **os_write_lock**) indicating the type of lock ObjectStore was requesting when the timeout occurred. **os_lock_type** is a top-level enumeration.

os_lock_timeout_exception::get_pids()

typedef os_unsigned_int32 os_pid ;
os_pid *get_pids();

Returns an array of integers indicating the process IDs of the processes preventing lock acquisition.

os_lock_timeout_exception::number_of_blockers()

os_int32 number_of_blockers();

Returns the number of processes preventing lock acquisition.

Sample TIX Exception-Handling Routine

```
#include <stdlib.h>
#include <ostore/except.hh>
#include <iostream.h>

/* ObjectStore macro DEFINE_EXCEPTION takes three arguments:
        The formal name of the exception
        The generic message [a literal string] printed when the
            exception is raised and there is no handler to catch it.
        The parent exception ( = 0 when no parent);
*/

DEFINE_EXCEPTION(illegal_input_char, "ILLEGAL INPUT
CHARACTER", 0);

main()
{

char c = ‘\0’;

while( c != ‘q’ && c != ‘Q’ )
    {
        cout << "Enter vowel: " ;
        cin >> c;

        TIX_HANDLE (illegal_input_char)
        switch(c)
        {
            case ‘q’:
            case ‘Q’:
                break;
            case ‘A’:
            case ‘a’:
            case ‘E’:
            case ‘e’:
            case ‘I’:
            case ‘O’:
            case ‘o’:
            case ‘U’:
            case ‘u’:
                cout << "You have entered character: " << c <<"\n";
                break;
           default:
                illegal_input_char.signal("");
                break;
        }

        TIX_EXCEPTION
            cout << "Only vowels are legal.  ";
        /* as many statements as you may choose to write are
        allowed here */
            cout << "Try again\n";
        TIX_END_HANDLE
        }

    c = ‘\0’;    /* reset c */

while( c != ‘q’ && c != ‘Q’ )
   {
        cout << "Enter consonant: " ;
        cin >> c;

        switch(c)
        {
            case ‘q’:
            case ‘Q’:
                break;
            case ‘A’:
            case ‘a’:
            case ‘E’:
            case ‘e’:
            case ‘I’:
            case ‘O’:
            case ‘o’:
            case ‘U’:
            case ‘u’:
                illegal_input_char.signal("\tInput character ‘%c’ not a
consonant\n", c);
        break;
        default:
        cout << "You have entered character: " << c << "\n";
        break;
        }
        }
    }
```

Appendix B
Predefined TIX Exceptions

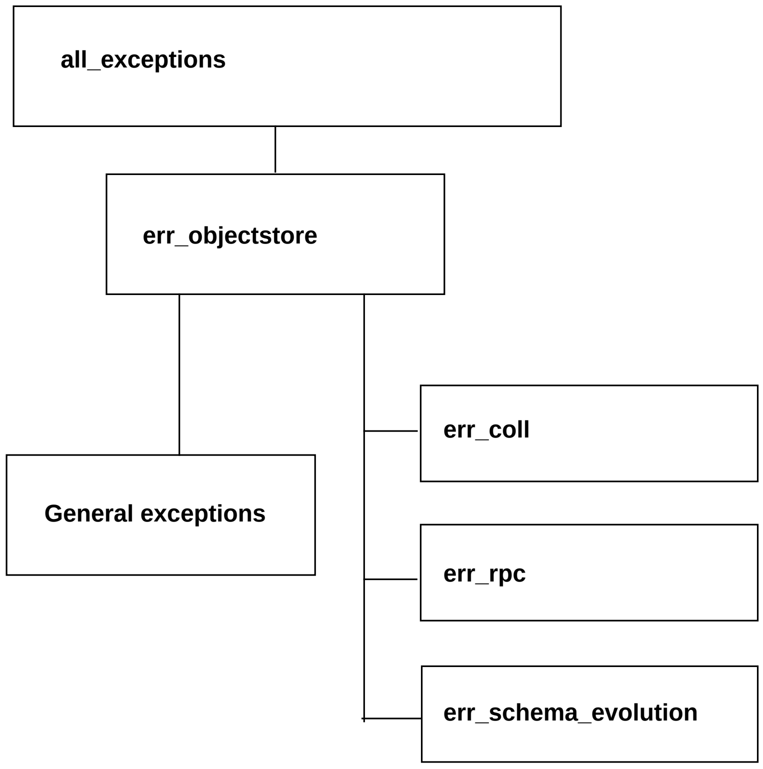
This section contains information on significant predefined exceptions. These exceptions are defined in **client.hh** and **ostore.h**, so they are automatically available to your programs.

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 object tree hierarchy

The hierarchy is arranged as follows:



- Every TIX exception is a descendent 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**.

General ObjectStore Exceptions

The following exceptions are all descended from **err_objectstore**.

err_abort_committed.

err_address_space_full. Address space is full.

err_alignment. A memory fault was signaled by the operating system, but rather than the usual access fault, this fault indicates that there was an alignment error. Often this means that there was an attempt to dereference through a pointer whose value is odd.

err_architecture_mismatch. The architecture of the database is not compatible with that of the application.

err_authentication_failure.

err_awaiting_recovery. The Server cannot perform this operation, ultimately because some Server is down and needs to be brought back up, specifically because it operates on a block that is tied up because of a two-phase commit that is still in progress because of the Server crash. Check the console messages to see explanatory messages from the two-phase commit recovery procedure.

err_beyond_segment_length. Several meanings, most commonly that the client sent a block number that was higher than the highest block in the segment.

err_broken_server_connection. This usually means the same thing as **err_network_failure**, and might mean network trouble, or that the Server has crashed. See the class **os_server**.

err_cache_manager_not_responding. The Server was unable to form a network connection to the Cache Manager process of your client host. You should see whether the Cache Manager process appears to be alive; if not, look in the operating system error log for an error message that might help explain what happened to the Cache Manager.

err_cannot_commit. ObjectStore tried to perform a commit involving updates to multiple servers (that is, a two-phase commit) and it cannot determine a common network for communications.

err_cannot_open_application_schema. This is signaled if you try to create or look up a database, but the application schema database for your application cannot be found.

err_checkpoint_aborted.

err_checkpoint_committed.

err_checkpoint_not_inner.

err_cluster_full.

err_cluster_mismatch.

err_commit_aborted.

err_commit_abort_only. Signaled when the user attempts to commit an abort-only transaction.

err_commit_with_children.

err_conflicting_failover_configuration.

err_connection_closed.

err_cursor_at_end.

err_cursor_not_at_object.

err_database_not_open. Signaled by the notification APIs.

err_database_transient. Signaled by the notification APIs.

err_cross_db_pointer. A pointer from one database to another was found. This happens only if external pointers are not allowed. See, for example, **os_database::allow_external_pointers()**.

err_database_exists. The database already exists. This can only happen from **mkdir**.

err_database_id_exists. The client requested the Server to create a new database with a certain database ID, but the Server found that a database with that database ID already exists. Ordinarily, the Server invents a new database ID, so this cannot happen. The error can occur when the client side specifies a particular database ID, which normally happens only when you are reloading a backup tape or using the **osdbimport** utility.

err_database_id_not_found. Several meanings, most commonly that the **database_handle** sent by the client was not valid.

err_database_is_deleted. The database has been deleted.

errr__database_lock_conflict.

err_database_not_found. The specified database was not found. For example, the application tried to look up **/a/b/c**, but the directory **/a/b** did not contain a database named **c**.

err_database_not_open. The database is not open.

err_database_transient. This operation is not allowed on the transient database.

err_database_wrong_version. Currently, there is only one version of database format, so this exception cannot be signaled. In the future, however, the format of databases might change. This exception is signaled if an executable file linked with an older version of ObjectStore, which cannot understand the new format, attempts to operate on a database in the new format.

err_datagram_error.

err_deadlock. A deadlock was detected. Note that ObjectStore transactions establish handlers for this exception.

err_deref_transient_pointer. An attempt was made to dereference a pointer to transient memory. The operating system signaled a virtual memory access fault, but the address of the fault was within the transient area of the address space, rather than the persistent area.

err_destroy_cluster_not_empty.

err_directory_exists. This is signaled by **objectstore::make_ directories** or **objectstore::mkdir** when you try to make a new directory on the Directory Manager, but a directory with that name already exists.

err_directory_not_empty. The directory was not empty, but the application tried to delete the directory. This can only happen from **mkdir.**

err_directory_not_found. The specified pathname contained a directory component that was not found.

err_discriminant_out_of_bounds. The value is larger than the number of alternatives for the union.

err_explicit_commit_stack_txn.

err_file_error.

err_file_not_db.

err_file_not_local. You have asked a Server to operate on a file database that is stored not on that Server host, but on some other host that the Server host has mounted using NFS. This should only be possible if you specify a Server explicitly for file databases, which is not the usual way file databases are used.

err_file_pathname. An entry point has checked a pathname and signaled that it is a file pathname rather than a rawfs pathname.

err_hardware. A memory fault was signaled by the operating system, but rather than the usual access fault, this fault indicates that there was a hardware error, such as a memory bus timeout.

err_inconsistent_db.

err_illegal_acquire_lock.

err_insufficient_virtual_memory. **malloc** space is used up.

err_internal. Several possible meanings: (1) The client did a **RELEASE_BLOCKS** on a block it does not own. (2) The client sent a block number that was higher than the highest block in the segment. (3) The **database_handle** sent by the client was not valid. (4) An operation was done on a segment that has been deleted, or is believed not to exist. Note that you cannot handle **err_internal**.

err_invalid_deletion. An argument that was not a pointer to a persistent object was passed to **operator delete**.

err_invalid_pathname. A syntactically invalid pathname was specified.

err_invalid_reference_assignment. The referent of an **os_ Reference_this_DB** is in a different database from the reference.

err_invalid_root_value. You have tried to store a transient or a cross-database pointer into a root.

err_license_limit. No connections are available, due to licensing limitations. This refers to the maximum number of connections, as specified by the Server password file.

err_link_not_found. Intrarawfs link was not found.

err_locator_file_id.

err_locator_read_only.

err_locator_syntax.

err_lock_timeout.

err_misc. Specifc to Notifications are as follows:

- The notification kind specified is reserved for use by ObjectStore.
- The notification queue size must be set before subscribing to notifications.
- The notification queue size must be greater than 0 but no larger than %d entries.
- Error connecting to the cache manager notification service: %s.
- Connection to cache manager was broken.

err_mvcc_nested.

err_net_cant_connect.

err_net_connect_timeout.

err_net_connection_refused..

err_net_host_down.

err_indigestible_pna.

err_net_no_server.

e**rr_network**.

err_network_failure. The network connection failed, possibly due to network trouble, or because the Server or Directory Manager crashed.

err_no_credentials. Access is not permitted; no credentials were presented.

err_no_query_trans.

err_no_rawfs. There is no rawfs on the Server.

err_no_service_on_net.

err_no_such_host. The specified host does not exist.

err_no_trans. There is no transaction in progress.

err_not_a_database. The pathname is not the pathname of a database. For example, the application tried to delete a database called **/a/b**, but **/a/b** was a directory.

err_not_a_directory. A pathname that was expected to be a directory is something else, possibly a database.

err_not_assigned. No database is assigned to the specified location. The application dereferenced a pointer that pointed into an area of virtual memory used for persistent objects, but which was not a valid location, because no database was mapped to the specified address. Usually this means that the application saved a pointer from a previous transaction and then used it in a subsequent transaction, which is not allowed.

err_null_pointer. An attempt was made to dereference a null (zero) pointer.

err_opened_read_only. An attempt was made to write a database that was opened as read-only.

err_operation_not_supported.

err_operator_new_args. Invalid arguments were specified for ObjectStore's **operator new**. Currently, this can happen only if you try to pass **0** to **operator new** as the *count* argument.

err_os_compaction. Invalid arguments to **objectstore::compact()**.

err_permission_denied. Permission to access this database was denied.

err_prepare_to_commit.

err_protocol_not_supported.

err_pvar_init_failure. An error occurred while trying to initialize a persistent variable. Currently, this can happen only if a persistent variable has no value yet, and there is no initializer on any declaration of the persistent variable.

err_pvar_type_mismatch. A persistent variable type mismatch was detected. This happens if two applications use a database; the first has a persistent variable with a particular name and type, and the second has a persistent variable with the same name but with a different type.

err_rawfs_not_upgraded. The rawfs is from an old release.

err_read_only_file_system. The file database is stored in a read-only file system.

err_reference_not_found. An attempt was made to resolve an object that has been deleted. This can happen to any safe reference, or to an unsafe reference when the target segment has been deleted.

err_reference_syntax. A string with improper syntax was passed to the **load()** member of a reference class.

err_reference_to_transient. The application attempted to create a safe reference to a transient location, which is not allowed.

err_restartable.

err_root_exists. A root with the specified name already exists. This occurs only if you create a root explicitly; it cannot happen as a result of implicit root creation by persistent variables.

err_schema_database. Attempt to use a closed database as a schema database, or attempt to use as a schema database a database that itself stored its schema remotely.

err_schema_validation_error. An error occurred during schema validation.

err_schema_validation_failed. The schema could not be validated.

err_segment_is_deleted. Several meanings, most commonly that an operation was done on a segment that has been deleted, or is believed not to exist.

err_server_address_in_use.

err_server_cannot_open_cache. The Server, running on the same host as the client, could not open the cache file in order to use shared memory transfers.

err_server_full. The Server has run out of disk space.

err_server_not_superuser. The Server is not running as the superuser.

err_server_refused_connection. The Server refused to make the requested connection.

err_server_restarted.

err_string_too_long. Notification string is longer than 16,383 characters.

err_too_many_cross_svr_links. Excessively long cross-Server link chain. The maximum depth of a cross-Server link chain is currently 15.

err_too_many_links. Too many levels of intrarawfs links.

err_too_many_retries. The transaction was retried more times than the value specified by **os_transaction::max_retries**.

err_trans. An operation was performed outside a transaction when it should have been within one, or an operation was performed within a transaction when it should have been outside any transaction. For example, an attempt was made to access persistent memory from outside any transaction.

err_trans_wrong_type. An attempt was made to start a nested transaction whose type (**update** or **read_only**) differs from the type of the transaction within which it is nested.

err_transient_pointer. A transient pointer was found in a database.

err_uninitialized. This error should not occur. It means the database is uninitialized, possibly because the application that created the database aborted before it finished running, leaving the database empty and not yet initialized. If this occurs, you should delete the database and recreate it.

err_unknown_pointer. While pointers were being relocated, ObjectStore came upon a pointer that did not seem to be a valid pointer to any object.

err_user_aborted_simple_authentication.

err_write_during_query. An attempt was made to write during a read-only transaction. *Write* refers not only to storing into persistent objects, but also to other operations that modify the database, such as creating segments.

err_write_permission_denied. Permission to write this database was denied.

Schema Evolution Exceptions

The following exceptions are all descended from **err_schema_ evolution**, which is a descendent of **err_objectstore**.

err_se_ambiguous_subtype_evolution. An attempt was made to evolve an object to a subtype in which it is not a unique supertype.

err_se_ambiguous_void_pointer. A void pointer was specified that could not be interpreted unambiguously. This is the case when there are nested data that occupy the same location before schema evolution, but different locations afterwards.

err_se_cannot_delete_class. The specified class cannot be deleted from the schema, since other classes still depend upon it.

err_se_cross_database_pointer. An attempt was made to resolve a cross-database pointer to a database that was not part of the set of databases being evolved.

err_se_deleted_component. A pointer was specified to the deleted component of a valid object.

err_se_deleted_object. A pointer was specified to a deleted object.

err_se_illegal_pointer. This is the base type for various illegal pointers.

err_se_invalid_subtype_evolution. An attempt was made to evolve an object to a type that was not a subtype.

err_se_pointer_type_mismatch.

err_se_transient_object. A pointer to a transient object was encountered.

err_se_unnecessary. The requested schema evolution was unnecessary; all the schemas under consideration are compatible.

Metaobject Protocol Exceptions

The following exceptions are descended from **err_mop**, which is a descendent of **err_objectstore**.

err_mop.

err_mop_forward_definition. An attempt was made to access information about a class for which only a forward definition exists, and the information required a full definition.

err_mop_illegal_cast. An attempt was made to cast a metaobject to an inappropriate type.

RPC Exceptions

The following exceptions are children of **err_rpc**. They are rarely signaled; only the most common ones are described in detail below.

err_rpc_cantrecv. The operating system signaled an error when ObjectStore tried to send data. The textual message from the operating system is included in the character-string report in the exception.

err_rpc_cantsend. The operating system signaled an error when ObjectStore tried to send data. The textual message from the operating system is included in the character-string report in the exception.

err_rpc_timedout. The Server or the Directory Manager timed out; that is, the client was waiting for a reply, and after a certain fixed amount of time, did not receive one. This can mean, among other things, that the Server or Directory Manager has crashed, that it is slow to respond, or that the network is not working properly.

err_rpc_auth_badcred.

err_rpc_auth_badverf.

err_rpc_autherror.

err_rpc_auth_failed.

err_rpc_auth_invalidresp.

err_rpc_auth_ok.

err_rpc_auth_rejectedcred.

err_rpc_auth_rejectedverf.

err_rpc_auth_tooweak.

err_rpc_cantdecodeargs.

err_rpc_cantdecoderes.

err_rpc_cantencodeargs.

err_rpc_failed.

err_rpc_pmapfailure.

err_rpc_procunavail.

err_rpc_prognotregistered.

err_rpc_progunavail.

err_rpc_progversmismatch.

err_rpc_success.

err_rpc_systemerror.

err_rpc_unknownhost.

err_rpc_unknownproto.

err_rpc_versmismatch.

Component Schema Exceptions

The following exceptions exist:

err_schema_not_loaded

"The program schema must be loaded before doing this operation"

err_schema_not_found "The specified program schema was not found"

err_bad_schema_info "The program schema info being loaded is invalid"

err_invalid_for_application_schema "This operation is invalid for the application schema"

err_cannot_open_DLL_schema "Unable to open DLL schema database"

err_transient_dope_damaged "Some objects' transient dope could be invalid"

err_DLL "Operating system DLL operation failed"

err_DLL_not_loaded "Shared library could not be loaded"

err_DLL_not_unloaded "Shared library could not be unloaded"

err_DLL_symbol_not_found "Symbol could not be found in shared library"

Index

A

abort()
  os_transaction, defined by 331, 380
abort_in_progress()
  objectstore, defined by 11
abort_only
  os_transaction, defined by 331, 379
abort_top_level()
  os_transaction, defined by 332, 380
abort-only transactions 379
aborts 380
access control 307
Access_modifier
  os_member, defined by 154
acquire_lock()
  objectstore, defined by 11
add_DLL_identifier()
  os_DLL_schema_info, defined by 130
add_missing_dispatch_table_handler()
  objectstore, defined by 14
after_begin
  os_transaction_hook, defined by 339
after_commit
  os_transaction_hook, defined by 339
allow_external_pointers mode 295
allow_external_pointers()
  os_database, defined by 72, 203, 209, 215, 220, 224, 231, 237, 242, 367
  os_segment, defined by 220, 224, 231, 237, 242, 295, 367
Anonymous_indirect
  os_type, defined by 342
Anonymous_union
  os_class_type, defined by 57
Application_schema
  os_schema, defined by 270
Array
  os_type, defined by 342
assign()
  os_notification, defined by 176
  os_subscription, defined by 319
augment_classes_to_be_recycled()
  os_schema_evolution, defined by 276
augment_classes_to_be_removed()
  os_schema_evolution, defined by 277
augment_post_evol_transformers()
  os_schema_evolution, defined by 277
augment_subtype_selectors()
  os_schema_evolution, defined by 278
authorization 307

B

basic_undo()
  basic_undo, defined by 563
basic_undo, the class
  ~basic_undo() 563

basic_undo() 563
exception 563
before_abort
os_transaction_hook, defined by 339
before_commit
os_transaction_hook, defined by 339
before_retry
os_transaction_hook, defined by 339
begin()
os_transaction, defined by 332
bind()
os_mop, defined by 170

C

change_array_length()
objectstore, defined by 14
change_database_reference()
os_database, defined by 73
change_mapping()
os_str_conv, defined by 314
change_schema_key()
os_database, defined by 73
Char
os_type, defined by 342
char_os_fetch() 439
char_os_store() 439
check_illegal_pointers mode 301
checkpoint
os_dbutil::svr_checkpoint() 119
os_transaction::checkpoint() 334
checkpoint()
os_transaction, defined by 334
checkpoint_in_progress()
os_transaction, defined by 335
chgrp()
os_dbutil, defined by 106
chmod()
os_dbutil, defined by 106
chown()
os_dbutil, defined by 106
Class
os_class_type, defined by 57
os_type, defined by 342
class library 7
class, system-supplied
nonparameterized
os_reference 209–214
os_reference_local 220–223
os_reference_protected 231–236
os_reference_protected_local 242–246
os_reference_this_DB 252–256
os_reference_transient 262–265
objectstore 11–43
os_access_modifier 44
os_address_space_marker 45–48
os_anonymous_indirect_type 49
os_app_schema 50
os_array_type 52–53
os_base_class 54–56
os_class_type 57–69
os_comp_schema 70
os_database 71–100
os_database_reference 101–102
os_database_root 103–104
os_database_schema 105
os_dbutil 106–127
os_DLL_finder 128–129
os_enum_type 135–136
os_enumerator_literal 137
os_evolve_subtype_fun_binding 138
os_failover_server 139–140
os_field_member_variable 141
os_function_type 142–143
os_indirect_type 144
os_instantiated_class_type 145
os_integral_type 146–147
os_literal 148–150
os_literal_template_actual_arg 151
os_lock_timeout_exception 152–153
os_member 154–158
os_member_function 159–163
os_member_namespace 164

os_member_type 165
os_member_variable 166–169
os_mop 170–173
os_named_indirect_type 174
os_namespace 175
os_notification 176–185
os_object_cluster 186–188
os_object_cursor 189–190
os_pathname_and_segment_number 191
os_pointer_literal 192
os_pointer_to_member_type 193
os_pointer_type 194
os_pragma 195
os_pvar 196–198
os_rawfs_entry 199–201
os_real_type 202
os_Reference 203–208
os_reference 209–214
os_Reference_local 215–219
os_reference_local 220–223
os_Reference_protected 224–230
os_reference_protected 231–236
os_Reference_protected_local 237–241
os_reference_protected_local 242–246
os_Reference_this_DB 247–251
os_reference_this_DB 252–256
os_Reference_transient 257–261
os_reference_transient 262–265
os_reference_type 266
os_relationship_member_variable 267
os_retain_address 268–269
os_schema 270–271
os_schema_evolution 272–288
os_schema_handle 289–291
os_schema_info 293–??, 293
os_segment 295–306
os_segment_access 307–310
os_server 311–313
os_str_conv 314–317
os_subscription 318–320
os_template 321–322
os_template_actual_arg 323–324
os_template_formal_arg 325–326
os_template_instantiation 327–328
os_template_type_formal 329
os_template_value_formal 330
os_transaction 331–338
os_transaction_hook 339–340
os_transformer_binding 341
os_type 342–353
os_type_template 354
os_type_template_actual_arg 355
os_type_type 356
os_typed_pointer_void 357
os_typespec 358–362
os_void_type 363
os_with_ mapped 364
parameterized
os_DLL_finder 128–129
os_Reference 203–208
os_Reference_local 215–219
os_Reference_protected 224–230
os_Reference_protected_ local 237–241
os_Reference_this_DB 247–251
os_Reference_transient 257–261
os_schema_handle 289–291
os_schema_info 293–??, 293
close()
os_database, defined by 74
close_all_server_connections()
os_dbutil, defined by 107
close_server_connection()
os_dbutil, defined by 107
clustering
specifying using **operator new()** 367
cmgr_remove_file()
os_dbutil, defined by 107
cmgr_shutdown()
os_dbutil, defined by 107
cmgr_stat()
os_dbutil, defined by 108

commit()
os_transaction, defined by 335
compact()
objectstore, defined by 15
compaction 15–17
compare_schemas()
os_dbutil, defined by 110
Compilation_schema
os_schema, defined by 270
concurrency control
locking 379
connection_is_broken()
os_server, defined by 312
convert()
os_str_conv, defined by 315
copy_classes()
os_mop, defined by 171
copy_database()
os_dbutil, defined by 110
create()
os_access_modifier, defined by 44
os_anonymous_indirect_type, defined by 49
os_array_type, defined by 52
os_base_class, defined by 54
os_class_type, defined by 57
os_database, defined by 74
os_enum_type, defined by 135
os_enumerator_literal, defined by 137
os_field_member_variable, defined by 141
os_function_type, defined by 142
os_instantiated_class_type, defined by 145
os_integral_type, defined by 146
os_literal_template_actual_arg, defined by 151
os_member_function, defined by 159
os_member_namespace, defined by 164
os_member_type, defined by 165
os_member_variable, defined by 166
os_named_indirect_type, defined by 174
os_namespace, defined by 175
os_pointer_literal, defined by 192
os_pointer_to_member_type, defined by 193
os_pointer_type, defined by 194
os_pragma, defined by 195
os_real_type, defined by 202
os_reference_type, defined by 266
os_template_instantiation, defined by 327
os_template_type_formal, defined by 329
os_template_value_formal, defined by 330
os_type_template, defined by 354
os_type_template_actual_arg, defined by 355
os_void_type, defined by 363
create_char()
os_literal, defined by 148
create_defaulted_char()
os_integral_type, defined by 146
create_double()
os_real_type, defined by 202
create_enum_literal()
os_literal, defined by 148
create_float()
os_real_type, defined by 202
create_int()
os_integral_type, defined by 146
create_long()
os_integral_type, defined by 146
create_long_double()
os_real_type, defined by 202
create_object_cluster()
os_segment, defined by 296
create_pointer_literal()
os_literal, defined by 148
create_root()
os_database, defined by 76

create_segment()
  os_database, defined by 76
create_short()
  os_integral_type, defined by 146
create_signed_char()
  os_integral_type, defined by 147
  os_literal, defined by 148
create_signed_int()
  os_literal, defined by 148
create_signed_long()
  os_literal, defined by 148
create_signed_short()
  os_literal, defined by 148
create_unsigned_char()
  os_integral_type, defined by 147
  os_literal, defined by 149
create_unsigned_int()
  os_literal, defined by 149
create_unsigned_long()
  os_literal, defined by 149
create_unsigned_short()
  os_literal, defined by 149
create_wchar_t()
  os_literal, defined by 149
cross-database pointers
  allow_external_pointers mode 295
  changing type of associated **os_database_reference** 73
  dereferencing 73, 296
  and instances of **os_database_reference** 101
  instances of **os_reference** as substitutes 209
  instances of **os_Reference_local** as substitutes 215
  instances of **os_reference_local** as substitutes 220
  instances of **os_Reference_protected** as substitutes 224
  instances of **os_reference_protected** as substitutes 231
  instances of **os_Reference_protected_local** as substitutes 237
  instances of **os_reference_protected_local** as substitutes 242
  and segment state 297
cross-transaction pointers
  disabling 31
  enabling 31
  instances of **os_Reference** as substitutes 203
  instances of **os_reference** as substitutes 209
  instances of **os_Reference_local** as substitutes 215
  instances of **os_reference_local** as substitutes 220
  instances of **os_Reference_protected** as substitutes 224
  instances of **os_reference_protected** as substitutes 231
  instances of **os_Reference_protected_local** as substitutes 237
  instances of **os_reference_protected_local** as substitutes 242
  instances of **os_Reference_this_DB** as substitutes 247
  instances of **os_reference_this_DB** as substitutes 252
  instances of **os_Reference_transient** as substitutes 257
  instances of **os_reference_transient** as substitutes 262
current()
  os_mop, defined by 171
  os_object_cursor, defined by 189

D

database access control 307
database permissions 307
database utility API 106

database_of()
  os_segment, defined by 296
database_pathname
  os_pathname_and_segment_number, defined by 191
Database_schema
  os_schema, defined by 270
databases
  and persistent data 71
  open count 71, 410
deadlock 380
decache()
  os_database, defined by 76
DECLARE_EXCEPTION(), the macro 558
declares_get_os_typespec_function()
  os_class_type, defined by 58
DEFINE_EXCEPTION(), the macro 557
defines_virtual_functions()
  os_class_type, defined by 58
defining_class()
  os_member, defined by 154
delete, the operator 366
delete_array()
  os_database_reference, defined by 101
deleted()
  os_Reference_protected, defined by 226
  os_reference_protected, defined by 232
  os_Reference_protected_local, defined by 238
  os_reference_protected_local, defined by 243
deregister_hook()
  os_transaction_hook, defined by 340
destroy()
  os_database, defined by 77
  os_object_cluster, defined by 186
  os_segment, defined by 296
destroy_always
  os_object_cluster, defined by 187
destroy_if_empty
  os_object_cluster, defined by 187
disconnect()
  os_server, defined by 312
discriminant functions 390
discriminant_swap_bytes()
  objectstore, defined by 17
disk_free()
  os_dbutil, defined by 110
DLL_loaded()
  os_DLL_schema_info, defined by 130, 131
DLL_unloaded()
  os_DLL_schema_info, defined by 131
  os_schema_handle, defined by 289
Double
  os_type, defined by 342
double_os_fetch() 440
double_os_store() 440
dump()
  os_Reference, defined by 205
  os_reference, defined by 210
  os_Reference_local, defined by 216
  os_reference_local, defined by 221
  os_Reference_protected, defined by 226
  os_reference_protected, defined by 232
  os_Reference_protected_local, defined by 238
  os_reference_protected_local, defined by 243
  os_Reference_this_DB, defined by 248
  os_reference_this_DB, defined by 253
  os_Reference_transient, defined by 258
  os_reference_transient, defined by 262

E

embedded_server_available()
  objectstore, defined by 18
enable_damaged_dope_repair()
  objectstore, defined by 18
enable_DLL_schema()
  objectstore, defined by 19
entry points 103

Enum
os_type, defined by 342
environment variables
OS_CACHE_FILE 34
OS_DEF_EXCEPT_ACTION 559, 560
OS_LOG_FILE 38
equal_DLL_identifiers()
os_DLL_finder, defined by 128
equal_signature()
os_function_type, defined by 142
err_abort_committed exception 571
err_address_space_full exception 571
err_alignment exception 571
err_architecture_mismatch exception 571
err_authentication_failure exception 571
err_awaiting_recovery exception 571
err_beyond_segment_length exception 571
err_broken_server_connection exception 571
handling with **os_server** 311
err_cache_manager_not_responding exception 571
err_cannot_commit exception 571
err_cannot_open_application_schema exception 572
err_checkpoint_aborted exception 572
err_checkpoint_committed exception 572
err_checkpoint_not_inner exception 572
err_cluster_full exception 369, 572
err_cluster_mismatch exception 572
err_commit_abort_only exception 572
err_commit_aborted exception 572
err_commit_with_children exception 572
err_conflicting_failover_configuration exception 572
err_connection_closed exception 572
err_cross_db_pointer exception 572
err_cursor_at_end exception 572
err_cursor_not_at_object exception 572
err_database_exists exception 572
err_database_id_exists exception 572
err_database_id_not_found exception 572
err_database_is_deleted exception 573
err_database_lock_conflict exception 573
err_database_not_found exception 573
err_database_not_open exception 572, 573
err_database_transient exception 572, 573
err_database_wrong_version exception 573
err_datagram_error exception 573
err_deadlock exception 573
err_deref_transient_pointer exception 573
err_destroy_cluster_not_empty exception 573
err_directory_exists exception 573
err_directory_not_empty exception 573
err_directory_not_found exception 573
err_discriminant_out_of_bounds exception 573
err_explicit_commit_stack_txn exception 574
err_file_error exception 574
err_file_not_db exception 574
err_file_not_local exception 574
err_file_pathname exception 574
err_hardware exception 574
err_illegal_acquire_lock exception 574
err_inconsistent_db exception 574
err_indigestible_pna exception 575
err_insufficient_virtual_memory exception 574
err_internal exception 574
err_invalid_deletion exception 366, 574
err_invalid_pathname exception 574
err_invalid_reference_assignment exception 574
err_invalid_root_value exception 574
err_license_limit exception 574
err_link_not_found exception 574
err_locator_file_id exception 575
err_locator_read_only exception 575
err_locator_syntax exception 575
err_lock_timeout exception 575

err_misc exception 368, 575
err_mop exception 579
err_mop_forward_definition exception 580
err_mop_illegal_cast exception 580
err_mvcc_nested exception 575
err_net_cant_connect exception 575
err_net_connect_timeout exception 575
err_net_connection_refused exception 575
err_net_host_down exception 575
err_net_no_server exception 575
err_network exception 575
err_network_failure exception 575
err_no_credentials exception 575
err_no_query_trans exception 575
err_no_rawfs exception 575
err_no_service_on_net exception 575
err_no_such_host exception 575
err_no_trans exception 576
err_not_a_database exception 576
err_not_a_directory exception 576
err_not_assigned exception 576
err_null_pointer exception 576
err_opened_read_only exception 576
err_operation_not_supported exception 576
err_operator_new_args exception 576
err_os_compaction exception 576
err_permission_denied exception 576
err_prepare_to_commit exception 576
err_protocol_not_supported exception 576
err_pvar_init_failure exception 576
err_pvar_type_mismatch exception 576
err_rawfs_not_upgraded exception 577
err_read_only_file_system exception 577
err_reference_not_found exception 577
err_reference_syntax exception 577
err_reference_to_transient exception 577
err_restartable exception 577
err_root_exists exception 577
err_rpc_auth_badcred exception 581
err_rpc_auth_badverf exception 581
err_rpc_auth_failed exception 581
err_rpc_auth_invalidresp exception 581
err_rpc_auth_ok exception 581
err_rpc_auth_rejectedcred exception 581
err_rpc_auth_rejectedverf exception 581
err_rpc_auth_tooweak exception 581
err_rpc_autherror exception 581
err_rpc_cantdecodeargs exception 581
err_rpc_cantdecoderes exception 581
err_rpc_cantencodeargs exception 581
err_rpc_cantrecv exception 581
err_rpc_cantsend exception 581
err_rpc_failed exception 581
err_rpc_pmapfailure exception 581
err_rpc_procunavail exception 582
err_rpc_prognotregistered exception 582
err_rpc_progunavail exception 582
err_rpc_progversmismatch exception 582
err_rpc_success exception 582
err_rpc_systemerror exception 582
err_rpc_timedout exception 581
err_rpc_unknownhost exception 582
err_rpc_unknownproto exception 582
err_rpc_versmismatch exception 582
err_schema_database exception 577
  schema database not open 89
err_schema_validation_error exception 577
err_schema_validation_failed exception 577
err_se_ambiguous_subtype_evolution exception 579
err_se_ambiguous_void_pointer exception 579
err_se_cannot_delete_class exception 579
err_se_cross_database_pointer exception 579
err_se_deleted_component exception 579
err_se_deleted_object exception 579
err_se_illegal_pointer exception 579
err_se_invalid_subtype_evolution exception 579
err_se_pointer_type_mismatch exception 579

err_se_transient_object exception 579
err_se_unnecessary exception 579
err_segment_is_deleted exception 577
err_server_address_in_use exception 577
err_server_cannot_open_cache exception 577
err_server_full exception 577
err_server_not_superuser exception 577
err_server_refused_connection exception 577
err_server_restarted exception 578
err_string_too_long exception 578
err_too_many_cross_svr_links exception 578
err_too_many_links exception 578
err_too_many_retries exception 578
err_trans exception 578
err_trans_wrong_type exception 578
  with nested transactions 379
err_transient_pointer exception 578
err_uninitialized exception 578
err_unknown_pointer exception 578
err_user_aborted_simple_authentication exception 578
err_write_during_query exception 578
err_write_permission_denied exception 578
evolve()
  os_schema_evolution, defined by 278
exception
  basic_undo, defined by 563
exception handlers
  establishing 383
exception_status
  os_transaction, defined by 335
exceptions
  action when no handler for 559, 560
  ancestor 560
  descendent 560
  destructor for automatic object 562
  err_abort_committed 571
  err_address_space_full 571
err_alignment 571
err_architecture_mismatch 571
err_authentication_failure 571
err_awaiting_recovery 571
err_beyond_segment_length 571
err_broken_server_connection 571
err_cache_manager_not_responding 571
err_cannot_commit 571
err_cannot_open_application_schema 572
err_checkpoint_aborted 572
err_checkpoint_committed 572
err_checkpoint_not_inner 572
err_cluster_full 572
err_cluster_mismatch 572
err_commit_abort_only 572
err_commit_aborted 572
err_commit_with_children 572
err_conflicting_failover_configuration 572
err_connection_closed 572
err_cross_db_pointer 572
err_cursor_at_end 572
err_cursor_not_at_object 572
err_database_exists 572
err_database_id_exists 572
err_database_id_not_found 572
err_database_is_deleted 573
err_database_lock_conflict 573
err_database_not_found 573
err_database_not_open 572, 573
err_database_transient 572, 573
err_database_wrong_version 573
err_datagram_error 573
err_deadlock 573
err_deref_transient_pointer 573
err_destroy_cluster_not_empty 573
err_directory_exists 573
err_directory_not_empty 573
err_directory_not_found 573
err_discriminant_out_of_bounds 573

err_explicit_commit_stack_txn 574
err_file_error 574
err_file_not_db 574
err_file_not_local 574
err_file_pathname 574
err_hardware 574
err_illegal_lock 574
err_inconsistent_db 574
err_indigestible_pna 575
err_insufficient_virtual_memory 574
err_internal 574
err_invalid_deletion 574
err_invalid_pathname 574
err_invalid_reference_assignment 574
err_invalid_root_value 574
err_license_limit 574
err_link_not_found 574
err_locator_file_id 575
err_locator_read_only 575
err_locator_syntax 575
err_lock_timeout 575
err_misc 575
err_mop 579
err_mop_forward_definition 580
err_mop_illegal_cast 580
err_mvcc_nested 575
err_net_cant_connect 575
err_net_connect_timeout 575
err_net_connection_refused 575
err_net_host_down 575
err_net_no_server 575
err_network 575
err_network_failure 575
err_no_credentials 575
err_no_query_trans 575
err_no_rawfs 575
err_no_service_on_net 575
err_no_such_host 575
err_no_trans 576
err_not_a_database 576
err_not_a_directory 576
err_not_assigned 576
err_null_pointer 576
err_opened_read_only 576
err_operation_not_supported 576
err_operator_new_args 576
err_os_compaction 576
err_permission_denied 576
err_prepare_to_commit 576
err_protocol_not_supported 576
err_pvar_init_failure 576
err_pvar_type_mismatch 576
err_rawfs_not_upgraded 577
err_read_only_file_system 577
err_reference_not_found 577
err_reference_syntax 577
err_reference_to_transient 577
err_restartable 577
err_root_exists 577
err_rpc_auth_badcred 581
err_rpc_auth_badverf 581
err_rpc_auth_failed 581
err_rpc_auth_invalidresp 581
err_rpc_auth_ok 581
err_rpc_auth_rejectedcred 581
err_rpc_auth_rejectedverf 581
err_rpc_auth_tooweak 581
err_rpc_autherror 581
err_rpc_cantdecodeargs 581
err_rpc_cantdecoderes 581
err_rpc_cantencodeargs 581
err_rpc_cantrecv 581
err_rpc_cantsend 581
err_rpc_failed 581
err_rpc_pmapfailure 581
err_rpc_procunavail 582
err_rpc_prognotregistered 582
err_rpc_progunavail 582
err_rpc_progversmismatch 582
err_rpc_success 582
err_rpc_systemerror 582
err_rpc_timedout 581

err_rpc_unknownhost 582
err_rpc_unknownproto 582
err_rpc_versmismatch 582
err_schema_database 577
err_schema_validation_error 577
err_schema_validation_failed 577
err_se_ambiguous_subtype_evolution 579
err_se_ambiguous_void_pointer 579
err_se_cannot_delete_class 579
err_se_cross_database_pointer 579
err_se_deleted_component 579
err_se_deleted_object 579
err_se_illegal_pointer 579
err_se_invalid_subtype_evolution 579
err_se_pointer_type_mismatch 579
err_se_transient_object 579
err_se_unnecessary 579
err_segment_is_deleted 577
err_server_address_in_use 577
err_server_cannot_open_cache 577
err_server_full 577
err_server_not_superuser 577
err_server_refused_connection 577
err_server_restarted 578
err_string_too_long 578
err_too_many_cross_svr_links 578
err_too_many_links 578
err_too_many_retries 578
err_trans 578
err_trans_wrong_type 578
err_transient_pointer 578
err_uninitialized 578
err_unknown_pointer 578
err_user_aborted_simple_authentication 578
err_write_during_query 578
err_write_permission_denied 578
general ObjectStore
handler 557
handling 556
metaobject protocol 579
ObjectStore internal errors 556
parent 560
predefined 569
remote procedure call (RPC) mechanism 581
schema evolution 579
signaling 559
variable 556

expand_global()
  os_dbutil, defined by 111

external_pointer_status()
  os_segment, defined by 297

F

fault handlers
  establishing 383

fetch policy
  setting 34

Field_variable
  os_member, defined by 154

find()
  os_database_root, defined by 103

find_base_class()
  os_class_type, defined by 58

find_DLL_schema()
  objectstore, defined by 19

find_member_variable()
  os_class_type, defined by 59

find_namespace()
  os_mop, defined by 171

find_root()
  os_database, defined by 78

find_type()
  os_mop, defined by 172
  os_schema, defined by 270

first()
  os_object_cursor, defined by 189

Float
  os_type, defined by 343

float_os_fetch() 440

float_os_store() 440
forget()
  os_Reference_protected, defined by 226
  os_reference_protected, defined by 233
  os_Reference_protected_local, defined by 239
  os_reference_protected_local, defined by 243
freeze_schema_key()
  os_database, defined by 78
Function
  os_member, defined by 154
  os_template, defined by 321
  os_type, defined by 343

G

get()
  os_app_schema, defined by 50
  os_comp_schema, defined by 70
  os_database_schema, defined by 105
  os_DLL_finder, defined by 128
  os_schema_handle, defined by 289
  os_schema_info, defined by 293
get_abs_path()
  os_rawfs_entry, defined by 199
get_access()
  os_base_class, defined by 54
  os_member, defined by 154
get_access_control()
  os_segment, defined by 297
get_access_of_get_os_typespec_function()
  os_class_type, defined by 59
get_alignment()
  os_type, defined by 343
get_all()
  os_schema_handle, defined by 289
get_all_databases()
  os_database, defined by 79
get_all_object_clusters()
  os_segment, defined by 297
get_all_roots()
  os_database, defined by 80
get_all_segments()
  os_database, defined by 80
get_all_segments_and_permissions()
  os_database, defined by 80
get_all_servers()
  objectstore, defined by 20
get_allocated_virtual_base_classes()
  os_class_type, defined by 59
get_application_info()
  os_database, defined by 81
  os_segment, defined by 297
get_application_names()
  os_lock_timeout_exception, defined by 152, 565
get_application_schema_handle()
  os_schema_handle, defined by 290
get_application_schema_pathname()
  objectstore, defined by 21
get_arg_list()
  os_function_type, defined by 142
get_arg_list_kind()
  os_function_type, defined by 143
get_args()
  os_template, defined by 321
  os_template_instantiation, defined by 327
get_as_intervals()
  objectstore, defined by 21
get_auto_open_mode()
  objectstore, defined by 21
get_autoload_DLLs_function()
  objectstore, defined by 22
get_base_classes()
  os_class_type, defined by 60
get_base_member()
  os_access_modifier, defined by 44
get_cache_file()
  objectstore, defined by 22

get_cache_size()
objectstore, defined by 22
get_call_linkage()
os_member_function, defined by 159
get_char()
os_typespec, defined by 358
get_char_value()
os_literal, defined by 149
get_check_illegal_pointers()
objectstore, defined by 22
os_database, defined by 81
os_segment, defined by 298
get_class()
os_base_class, defined by 54
get_class_kind()
os_class_type, defined by 60
get_classes()
os_schema, defined by 270
get_client_name()
os_dbutil, defined by 111
get_comment()
os_segment, defined by 298
get_converted_size()
os_str_conv, defined by 315
get_creation_time()
os_rawfs_entry, defined by 199
get_current()
os_address_space_marker, defined by 46
os_transaction, defined by 335
tix_handler, defined by 561
get_database()
os_notification, defined by 176
os_Reference, defined by 205
os_reference, defined by 211
os_Reference_protected, defined by 227
os_reference_protected, defined by 233
os_Reference_this_DB, defined by 248
os_reference_this_DB, defined by 253
os_subscription, defined by 319
get_database_key()
os_Reference, defined by 205
os_reference, defined by 211
os_Reference_local, defined by 216
os_reference_local, defined by 221
os_Reference_protected, defined by 227
os_reference_protected, defined by 233
os_Reference_protected_local, defined by 239
os_reference_protected_local, defined by 243
os_Reference_this_DB, defined by 249
os_reference_this_DB, defined by 253
os_Reference_transient, defined by 258
os_reference_transient, defined by 263
get_database_references()
os_database, defined by 81
os_segment, defined by 298
get_databases()
os_server, defined by 312
get_default()
os_segment_access, defined by 307
get_default_check_illegal_pointers()
os_database, defined by 81
get_default_lock_whole_segment()
os_database, defined by 81
get_default_null_illegal_pointers()
os_database, defined by 82
get_default_segment()
os_database, defined by 82
get_default_segment_size()
os_database, defined by 82
get_defining_class()
os_member, defined by 155
get_dirman_host_name()
os_database, defined by 82
get_dispatch_table_pointer_offset()
os_class_type, defined by 60
get_DLL_identifiers()
os_schema_handle, defined by 290
os_schema_info, defined by 293

get_double()
os_typespec, defined by 358
get_element_type()
os_array_type, defined by 52
get_enclosing_class()
os_type, defined by 343
get_enclosing_namespace()
os_namespace, defined by 175
get_enum_literal()
os_literal, defined by 149
get_enumerator()
os_enum_type, defined by 135
get_enumerators()
os_enum_type, defined by 135
get_evolved_address()
os_schema_evolution, defined by 283
get_evolved_object()
os_schema_evolution, defined by 283
get_exception()
tix_handler, defined by 561
get_failure_classes()
os_mop, defined by 172
get_fault_addr()
os_lock_timeout_exception, defined by 152, 565
_get_fd()
os_notification, defined by 177
get_fetch_policy()
os_database, defined by 83
os_segment, defined by 298
get_file_host_name()
os_database, defined by 83
get_float()
os_typespec, defined by 358
get_for_update()
os_database_schema, defined by 105
get_function_kind()
os_member_function, defined by 159
get_group_name()
os_rawfs_entry, defined by 199
get_host_name()
os_database, defined by 83
os_server, defined by 312
get_hostnames()
os_lock_timeout_exception, defined by 152, 565
get_id()
os_database, defined by 83
get_ignore_illegal_pointers()
os_schema_evolution, defined by 283
get_incremental_schema_installation()
objectstore, defined by 22
os_database, defined by 83
get_indirect_virtual_base_classes()
os_class_type, defined by 61
get_info()
os_object_cluster, defined by 187
get_instantiation()
os_instantiated_class_type, defined by 145
get_int()
os_typespec, defined by 359
get_kind()
os_literal, defined by 149
os_member, defined by 155
os_notification, defined by 177
os_schema, defined by 270
os_template, defined by 321
os_template_actual_arg, defined by 323
os_template_formal_arg, defined by 325
os_type, defined by 343
get_kind_string()
os_type, defined by 343
get_level()
os_address_space_marker, defined by 46
get_link_host()
os_rawfs_entry, defined by 199
get_link_path()
os_rawfs_entry, defined by 199

get_literal()
  os_literal_template_actual_arg, defined by 151
get_locator_file()
  objectstore, defined by 22
get_lock_status()
  objectstore, defined by 23
get_lock_type()
  os_lock_timeout_exception, defined by 153, 565
get_lock_whole_segment()
  os_database, defined by 83
  os_segment, defined by 298
get_log_file()
  objectstore, defined by 23
get_logical_server_hostnamel()
  os_failover_server, defined by 139
get_long()
  os_typespec, defined by 359
get_long_double()
  os_typespec, defined by 359
get_max_retries()
  os_transaction, defined by 336
get_members()
  os_class_type, defined by 61
  os_namespace, defined by 175
get_most_derived_class()
  os_class_type, defined by 61
get_n_databases()
  os_database, defined by 84
  os_server, defined by 313
get_n_object_clusters()
  os_segment, defined by 297, 299
get_n_roots()
  os_database, defined by 80, 84
get_n_sectors()
  os_rawfs_entry, defined by 199
get_n_segments()
  os_database, defined by 80, 84
get_n_servers()
  objectstore, defined by 23
get_name()
  os_class_type, defined by 64
  os_database_reference, defined by 101
  os_database_root, defined by 103
  os_enum_type, defined by 135
  os_enumerator_literal, defined by 137
  os_member_function, defined by 160
  os_member_variable, defined by 166
  os_named_indirect_type, defined by 174
  os_namespace, defined by 175
  os_pointer_literal, defined by 192
  os_rawfs_entry, defined by 199
  os_template_formal_arg, defined by 325
  os_transaction, defined by 336
  os_typespec, defined by 359
get_namespace()
  os_member_namespace, defined by 164
get_neutralized_classes()
  os_mop, defined by 172
get_next()
  os_address_space_marker, defined by 46
get_null_illegal_pointers()
  objectstore, defined by 23
  os_segment, defined by 299
get_number()
  os_segment, defined by 299
get_number_of_elements()
  os_array_type, defined by 52
get_object_range()
  objectstore, defined by 23
get_offset()
  os_base_class, defined by 55
  os_field_member_variable, defined by 141
  os_member_variable, defined by 167
get_online_server_hostname()
  os_failover_server, defined by 139

get_open_database()
os_Reference, defined by 205
os_reference, defined by 211
os_Reference_protected, defined by 227
os_reference_protected, defined by 233
get_opt_cache_lock_mode()
objectstore, defined by 24
os_database, defined by 84
get_os_typespec()
os_Reference, defined by 206
os_reference, defined by 211
os_Reference_local, defined by 217
os_reference_local, defined by 221
os_Reference_protected, defined by 227
os_reference_protected, defined by 233
os_Reference_protected_local, defined by 239
os_reference_protected_local, defined by 243
os_Reference_this_DB, defined by 249
os_reference_this_DB, defined by 253
get_page_size()
objectstore, defined by 24
get_parent()
os_transaction, defined by 336
get_pathname()
os_database, defined by 84
get_permission()
os_rawfs_entry, defined by 200
get_pids()
os_lock_timeout_exception, defined by 153, 565
get_pointer()
os_typespec, defined by 359
get_pointer_literal()
os_literal, defined by 150
get_pointer_numbers()
objectstore, defined by 24
get_pragmas()
os_class_type, defined by 64
os_enum_type, defined by 136
get_previous()
os_address_space_marker, defined by 46
get_primary_group()
os_segment_access, defined by 308
get_prms_are_in_standard_format()
os_database, defined by 85
get_readlock_timeout()
objectstore, defined by 24
os_database, defined by 85
os_segment, defined by 299
get_reconnect_retry_interval()
os_failover_server, defined by 139
get_reconnect_timeout()
os_failover_server, defined by 139
get_reference()
os_notification, defined by 177
get_related_class()
os_relationship_member_variable, defined by 267
get_related_member()
os_relationship_member_variable, defined by 267
get_relative_directory()
os_database, defined by 85
get_report()
tix_handler, defined by 561
get_report0()
tix_handler, defined by 562
get_required_DLL_identifiers()
os_database, defined by 85
get_retain_persistent_addresses()
objectstore, defined by 25
get_return_type()
os_function_type, defined by 143
get_schema_database()
os_database, defined by 86
os_schema_handle, defined by 290
get_schema_database_pathname()
os_schema_handle, defined by 290
os_schema_info, defined by 293

get_schema_info()
os_schema_handle, defined by 291
get_scope()
os_transaction, defined by 336
get_sector_size()
os_database, defined by 86
os_dbutil, defined by 112
get_segment()
os_database, defined by 86
get_server_host()
os_rawfs_entry, defined by 200
get_short()
os_typespec, defined by 360
get_signed_char()
os_typespec, defined by 360
get_signed_int()
os_typespec, defined by 360
get_signed_integral_value()
os_literal, defined by 150
get_signed_long()
os_typespec, defined by 360
get_signed_short()
os_typespec, defined by 361
get_simple_auth_ui()
objectstore, defined by 25
get_size()
os_base_class, defined by 55
os_field_member_variable, defined by 141
os_member_variable, defined by 167
os_type, defined by 344
get_size_as_base()
os_class_type, defined by 64
get_source_position()
os_class_type, defined by 64
os_enum_type, defined by 136
os_member_function, defined by 160
os_member_variable, defined by 167
os_named_indirect_type, defined by 174
get_status()
os_schema_handle, defined by 291
get_storage_class()
os_member_variable, defined by 167
get_string()
os_notification, defined by 177
os_pragma, defined by 195
os_type, defined by 344
get_target_class()
os_pointer_to_member_type, defined by 193
get_target_type()
os_anonymous_indirect_type, defined by 49
os_indirect_type, defined by 144
os_pointer_type, defined by 194
get_template()
os_template_instantiation, defined by 327
get_thread_locking()
objectstore, defined by 25
get_transient_database()
os_database, defined by 86
get_transient_delete_function()
objectstore, defined by 25
get_transient_schema()
os_mop, defined by 172
get_transient_segment()
os_segment, defined by 299
get_type()
os_literal, defined by 150
os_member_function, defined by 160
os_member_type, defined by 165
os_member_variable, defined by 166
os_pointer_literal, defined by 192
os_rawfs_entry, defined by 200
os_template_value_formal, defined by 330
os_transaction, defined by 336
os_type_template, defined by 354
os_type_template_actual_arg, defined by 355
os_typed_pointer_void, defined by 357

get_typespec()
  os_database_root, defined by 103
get_unevolved_address()
  os_schema_evolution, defined by 283
get_unevolved_object()
  os_schema_evolution, defined by 284
get_unsigned_char()
  os_typespec, defined by 361
get_unsigned_int()
  os_typespec, defined by 361
get_unsigned_integral_value()
  os_literal, defined by 150
get_unsigned_long()
  os_typespec, defined by 361
get_unsigned_short()
  os_typespec, defined by 361
get_user_name()
  os_rawfs_entry, defined by 200
get_value()
  os_database_root, defined by 103
  os_enumerator_literal, defined by 137
  tix_handler, defined by 560, 562
get_virtual_base_class_pointer_offset()
  os_base_class, defined by 55
get_wchar_t_value()
  os_literal, defined by 150
get_work_database()
  os_schema_evolution, defined by 284
get_writelock_timeout()
  objectstore, defined by 25
  os_database, defined by 86
  os_segment, defined by 299
global functions
  operator delete() 366
  operator delete[]() 366
  operator new() 367
  os_fetch() 370
  os_fetch_address() 373
  os_store() 374
  system-supplied 365

H

handler, exception 556
has_constructor()
  os_class_type, defined by 65
has_destructor()
  os_class_type, defined by 65
has_dispatch_table()
  os_class_type, defined by 65
hash()
  os_Reference, defined by 206
  os_reference, defined by 211
  os_Reference_local, defined by 217
  os_reference_local, defined by 221
  os_Reference_protected, defined by 227
  os_reference_protected, defined by 234
  os_Reference_protected_local, defined by 239
  os_reference_protected_local, defined by 244
  os_Reference_this_DB, defined by 249
  os_reference_this_DB, defined by 253
  os_Reference_transient, defined by 259
  os_reference_transient, defined by 263
hidden_write()
  objectstore, defined by 26

I

ignore_locator_file()
  objectstore, defined by 26
illegal pointers
  check_illegal_pointers mode 301
init_int()
  os_pvar, defined by 197, 198
init_long()
  os_pvar, defined by 197, 198
init_pointer()
  os_pvar, defined by 197, 198
initial segment 82, 94

initialize()
  objectstore, defined by 26
  os_dbutil, defined by 112
  os_mop, defined by 172
initialize_object_metadata()
  os_mop, defined by 172
insert()
  os_dynamic_extent, defined by 133
insert_required_DLL_identifier()
  os_database, defined by 86
insert_required_DLL_identifiers()
  os_database, defined by 87
  os_schema_handle, defined by 291
install()
  os_database_schema, defined by 105
Instantiated_class
  os_type, defined by 344
int_os_fetch() 440
int_os_store() 440
Integer
  os_type, defined by 344
intradatabase pointers
  instances of **os_Reference_this_DB** as substitutes 247
  instances of **os_reference_this_DB** as substitutes 252
introduces_virtual_functions()
  os_class_type, defined by 65
is_aborted()
  os_transaction, defined by 336
is_abstract()
  os_class_type, defined by 65
is_class_type()
  os_type, defined by 344
is_committed()
  os_transaction, defined by 336
is_const()
  os_anonymous_indirect_type, defined by 49
  os_member_function, defined by 160
  os_type, defined by 344
is_damaged_dope_repair_enabled()
  objectstore, defined by 27
is_db()
  os_rawfs_entry, defined by 200
is_deleted()
  os_segment, defined by 300
is_dir()
  os_rawfs_entry, defined by 200
is_DLL_schema_enabled()
  objectstore, defined by 27
is_empty()
  os_object_cluster, defined by 187
  os_segment, defined by 300
is_failover()
  os_server, defined by 313
is_field()
  os_member_variable, defined by 167
is_forward_definition()
  os_class_type, defined by 65
is_indirect_type()
  os_type, defined by 344
is_inline()
  os_member_function, defined by 160
is_integral_type()
  os_type, defined by 345
is_link()
  os_rawfs_entry, defined by 200
is_lock_contention()
  objectstore, defined by 27
is_open()
  os_database, defined by 87
is_open_mvcc()
  os_database, defined by 87
is_open_read_only()
  os_database, defined by 87
is_overloaded()
  os_member_function, defined by 161
is_persistent()
  objectstore, defined by 28
  os_class_type, defined by 65
  os_member_variable, defined by 168

is_pointer_type()
  os_type, defined by 344
is_pure_virtual()
  os_member_function, defined by 161
is_real_type()
  os_type, defined by 345
is_recognized()
  os_pragma, defined by 195
is_signed()
  os_integral_type, defined by 147
is_static()
  os_member_function, defined by 161
  os_member_variable, defined by 167
is_template_class()
  os_class_type, defined by 66
is_unspecified()
  os_member, defined by 155
  os_template, defined by 321
  os_type, defined by 345
is_virtual()
  os_base_class, defined by 55
  os_member_function, defined by 161
is_volatile()
  os_anonymous_indirect_type, defined by 49
  os_member_function, defined by 161
  os_type, defined by 345
is_writable()
  os_database, defined by 87

K

Known
  os_function_type, defined by 143

L

list_directory()
  os_dbutil, defined by 112
load()
  os_Reference, defined by 206
  os_reference, defined by 211
  os_Reference_local, defined by 217
  os_reference_local, defined by 221
  os_Reference_protected, defined by 227
  os_reference_protected, defined by 234
  os_Reference_protected_local, defined by 239
  os_reference_protected_local, defined by 244
  os_Reference_this_DB, defined by 249
  os_reference_this_DB, defined by 254
  os_Reference_transient, defined by 259
  os_reference_transient, defined by 263
load_DLL()
  objectstore, defined by 28
lock_as_used
  objectstore, defined by 29, 298, 303
lock_into_cache()
  os_segment, defined by 300
lock_segment_read
  objectstore, defined by 29, 298, 303
lock_segment_write
  objectstore, defined by 29, 298, 303
locking
  read-locking 379
  write-locking 379
Long_double
  os_type, defined by 345
long_double_os_fetch() 441
long_double_os_store() 441
long_os_fetch() 441
long_os_store() 441
lookup()
  os_database, defined by 88
lookup_DLL_symbol()
  objectstore, defined by 29

M

macro, system-supplied
  OS_BEGIN_TXN() 378
  OS_BEGIN_TXN_NAMED() 378
  OS_END_FAULT_HANDLER 383

OS_END_TXN() 378
OS_ESTABLISH_FAULT_HANDLER 383
OS_MARK_QUERY_FUNCTION() 385
OS_MARK_SCHEMA_TYPESPEC() 386
make_link()
os_dbutil, defined by 112
-make_reachable_source_classes_ persistent option to **ossg**
classes to mark 386
max_retries
os_transaction, defined by 380
metaobject protocol exceptions 579
mkdir()
os_dbutil, defined by 113
more()
os_object_cursor, defined by 190
multiversion concurrency control
open_mvcc() 90
MVCC
See multiversion concurrency control

N

Named_indirect
os_type, defined by 346
Namespace
os_member, defined by 155
nested transactions
and locking 379
network_servers_available()
objectstore, defined by 29
new, the operator 366
next()
os_object_cursor, defined by 190
no_access
os_segment_access, defined by 308
notify_immediate()
os_notification, defined by 178
notify_on_commit()
os_notification, defined by 179
number_of_blockers()
os_lock_timeout_exception, defined by 153, 565

O

ObjectStore exceptions
See exceptions
objectstore, the class 11–43
abort_in_progress() 11
acquire_lock() 11
add_missing_dispatch_table_ handler() 14
change_array_length() 14
compact() 15
discriminant_swap_bytes() 17
embedded_server_available() 18
enable_damaged_dope_repair() 18
enable_DLL_schema() 19
find_DLL_schema() 19
get_all_servers() 20
get_application_schema_pathname() 21
get_as_intervals() 21
get_auto_open_mode() 21
get_autoload_DLLs_function() 22
get_cache_file() 22
get_cache_size() 22
get_check_illegal_pointers() 22
get_incremental_schema_ installation() 22
get_locator_file() 22
get_lock_status() 23
get_log_file() 23
get_n_servers() 23
get_null_illegal_pointers() 23
get_object_range() 23
get_opt_cache_lock_mode() 24
get_page_size() 24
get_pointer_numbers() 24
get_readlock_timeout() 24
get_retain_persistent_addresses() 25
get_simple_auth_ui() 25

get_thread_locking() 25
get_transient_delete_function() 25
get_writelock_timeout() 25
hidden_write() 26
ignore_locator_file() 26
initialize() 26
is_damaged_dope_repair_enabled() 27
is_DLL_schema_enabled() 27
is_lock_contention() 27
is_persistent() 28
load_DLL() 28
lock_as_used 29, 298, 303
lock_segment_read 29, 298, 303
lock_segment_write 29, 298, 303
lookup_DLL_symbol() 29
network_servers_available() 29
propagate_log() 30
release_maintenance() 30
release_major() 30
release_minor() 30
release_name() 31
release_persistent_addresses() 31
retain_persistent_addresses() 31, 203, 209, 215, 220, 224, 231, 237, 242
return_all_pages() 31
return_memory() 31
set_always_ignore_illegal_pointers() 32
set_always_null_illegal_pointers() 32
set_application_schema_pathname() 32
set_auto_open_mode() 33
set_autoload_DLLs_function() 32
set_cache_file() 34
set_cache_size() 35
set_check_illegal_pointers() 35
set_client_name() 35
set_commseg_size() 35
set_current_schema_key() 36
set_eviction_batch_size() 36
set_eviction_pool_size() 37
set_handle_transient_faults() 37
set_incremental_schema_installation() 37
set_locator_file() 37
set_log_file() 38
set_mapped_communications() 38
set_null_illegal_pointers() 38
set_opt_cache_lock_mode() 39
set_readlock_timeout() 39
set_reserve_as_mode() 39
set_simple_auth_ui() 39
set_thread_locking() 40
set_transaction_priority() 41
set_transient_delete_function() 41
set_writelock_timeout() 42
shutdown() 42
unload_DLL() 43
which_product() 43

objectstore_acquire_lock() 402
objectstore_delete() 397, 402
objectstore_delete_function() 405
objectstore_disc_swap_bytes() 402
objectstore_exception()
  objectstore_exception, defined by 559
objectstore_exception, the class
  objectstore_exception() 559
  signal() 559
  vsignal() 560
objectstore_full_initialize() 402
objectstore_get_all_servers() 403
objectstore_get_application_schema_pathname() 403
objectstore_get_auto_open_mode() 403
objectstore_get_auto_open_read_whole_segment_mode() 403
objectstore_get_cache_size() 403
objectstore_get_incremental_schema_installation() 403
objectstore_get_lock_status() 403
objectstore_get_n_servers() 404
objectstore_get_null_illegal_pointers() 404
objectstore_get_object_range() 404
objectstore_get_object_total_range() 404

objectstore_get_opt_cache_lock_mode() 404
objectstore_get_page_size() 404
objectstore_get_pointer_numbers() 404
objectstore_get_readlock_timeout() 405
objectstore_get_retain_persistent_addresses() 405
objectstore_get_simple_auth_ui() 405
objectstore_get_writelock_timeout() 405
objectstore_hidden_write() 405
objectstore_initialize() 397
objectstore_is_lock_contention() 405
objectstore_is_persistent() 405
objectstore_realloc 400
objectstore_realloc() 406
objectstore_retain_persistent_addresses() 406
objectstore_return_all_pages() 406
objectstore_return_memory() 406
objectstore_set_aid() 406
objectstore_set_always_ignore_illegal_pointers() 406
objectstore_set_always_null_illegal_pointers() 406
objectstore_set_application_schema_pathname() 407
objectstore_set_auto_open_read_whole_segment_mode() 407
objectstore_set_cache_size() 407
objectstore_set_client_name() 407
objectstore_set_commseg_size() 407
objectstore_set_current_schema_key() 407
objectstore_set_eviction_batch_size() 408
objectstore_set_eviction_pool_size() 408
objectstore_set_handle_transient_faults() 408
objectstore_set_incremental_schema_installation() 408
objectstore_set_mapped_communications() 408
objectstore_set_null_illegal_pointers() 408
objectstore_set_opt_cache_lock_mode() 408
objectstore_set_readlock_timeout() 408
objectstore_set_reserve_as_mode() 409
objectstore_set_simple_auth_ui() 409
objectstore_set_transient_delete_function() 409
objectstore_set_writelock_timeout() 409
of()
  os_address_space_marker, defined by 47
  os_database, defined by 88
  os_object_cluster, defined by 187
  os_segment, defined by 300
open count, database 71, 410
open()
  os_database, defined by 88
open_mvcc()
  os_database, defined by 89
operator !()
  os_Reference, defined by 207
  os_Reference_protected, defined by 228
  os_reference_protected, defined by 235
  os_Reference_protected_local, defined by 240
  os_reference_protected_local, defined by 244
operator !=()
  os_Reference, defined by 207
  os_reference, defined by 212
  os_Reference_local, defined by 217
  os_reference_local, defined by 222
  os_Reference_protected, defined by 229
  os_reference_protected, defined by 235
  os_Reference_protected_local, defined by 240
  os_reference_protected_local, defined by 244
  os_Reference_this_DB, defined by 250
  os_reference_this_DB, defined by 254
  os_Reference_transient, defined by 260

os_reference_transient, defined by 264
operator <()
os_Reference, defined by 207
os_reference, defined by 213
os_Reference_local, defined by 218
os_reference_local, defined by 222
os_Reference_protected, defined by 229
os_reference_protected, defined by 235
os_Reference_protected_local, defined by 240
os_reference_protected_local, defined by 245
os_Reference_this_DB, defined by 250
os_reference_this_DB, defined by 255
os_Reference_transient, defined by 260
os_reference_transient, defined by 264
operator <=()
os_Reference, defined by 208
os_reference, defined by 213
os_Reference_local, defined by 218
os_reference_local, defined by 223
os_Reference_protected, defined by 229
os_reference_protected, defined by 235
os_Reference_protected_local, defined by 241
os_reference_protected_local, defined by 245
os_Reference_this_DB, defined by 251
os_reference_this_DB, defined by 255
os_Reference_transient, defined by 261
os_reference_transient, defined by 265
operator =()
os_rawfs_entry, defined by 201
os_Reference, defined by 206
os_reference, defined by 212
os_Reference_local, defined by 217
os_reference_local, defined by 221
os_Reference_protected, defined by 228
os_reference_protected, defined by 234
os_Reference_protected_local, defined by 239
os_reference_protected_local, defined by 244
os_Reference_this_DB, defined by 250
os_reference_this_DB, defined by 254
os_Reference_transient, defined by 259
os_reference_transient, defined by 264
os_segment_access, defined by 308
operator ==()
os_database_reference, defined by 101
os_Reference, defined by 207
os_reference, defined by 212
os_Reference_local, defined by 217
os_reference_local, defined by 222
os_Reference_protected, defined by 228
os_reference_protected, defined by 234
os_Reference_protected_local, defined by 240
os_reference_protected_local, defined by 244
os_Reference_this_DB, defined by 250
os_reference_this_DB, defined by 254
os_Reference_transient, defined by 260
os_reference_transient, defined by 264
os_typespec, defined by 362
operator >()
os_Reference, defined by 207
os_reference, defined by 213
os_Reference_local, defined by 218
os_reference_local, defined by 222
os_Reference_protected, defined by 229
os_reference_protected, defined by 235
os_Reference_protected_local, defined by 240
os_reference_protected_local, defined by 245
os_Reference_this_DB, defined by 250
os_reference_this_DB, defined by 255
os_Reference_transient, defined by 260
os_reference_transient, defined by 264
operator –>()
os_Reference, defined by 206

os_Reference_protected, defined by 228
os_Reference_this_DB, defined by 250
os_Reference_transient, defined by 259
operator >=()
os_Reference, defined by 208
os_reference, defined by 213
os_Reference_local, defined by 218
os_reference_local, defined by 222
os_Reference_protected, defined by 229
os_reference_protected, defined by 235
os_Reference_protected_local, defined by 241
os_reference_protected_local, defined by 245
os_Reference_this_DB, defined by 251
os_reference_this_DB, defined by 255
os_Reference_transient, defined by 260
os_reference_transient, defined by 265
operator const os_access_modifier&()
os_member, defined by 155
operator const os_anonymous_indirect_type&()
os_type, defined by 346
operator const os_app_schema&()
os_schema, defined by 270
operator const os_array_type&()
os_type, defined by 346
operator const os_class_type&()
os_type, defined by 346
operator const os_comp_schema&()
os_schema, defined by 271
operator const os_database_schema&()
os_schema, defined by 271
operator const os_enum_type&()
os_type, defined by 346
operator const os_field_member_variable&()
os_member, defined by 156
os_member_variable, defined by 168
operator const os_function_type&()
os_type, defined by 346
operator const os_instantiated_class_type&()
os_type, defined by 346
operator const os_integral_type&()
os_type, defined by 347
operator const os_literal_template_actual_arg&()
os_template_actual_arg, defined by 323
operator const os_member_function&()
os_member, defined by 156
operator const os_member_type&()
os_member, defined by 156
operator const os_member_variable&()
os_member, defined by 156
operator const os_named_indirect_type&()
os_type, defined by 347
operator const os_pointer_to_member_type&()
os_type, defined by 347
operator const os_pointer_type&()
os_type, defined by 347
operator const os_real_type&()
os_type, defined by 347
operator const os_reference_type&()
os_type, defined by 347
operator const os_relationship_member_variable&()
os_member, defined by 156
os_member_variable, defined by 168
operator const os_template_type_formal&()
os_template_formal_arg, defined by 325
operator const os_template_value_formal&()
os_template_formal_arg, defined by 325
operator const os_type_template&()
os_template, defined by 322
operator const os_type_template_actual_arg&()
os_template_actual_arg, defined by 323
operator const os_type_type&()
os_type, defined by 347

operator const os_void_type&()
os_type, defined by 348
operator delete(), the global function 366
operator delete[]() 366
operator new(), the global function 367
operator os_access_modifier&()
os_member, defined by 156
operator os_anonymous_indirect_type&()
os_type, defined by 348
operator os_app_schema&()
os_schema, defined by 271
operator os_array_type&()
os_type, defined by 348
operator os_class_type&()
os_type, defined by 348
operator os_comp_schema&()
os_schema, defined by 271
operator os_database_schema&()
os_schema, defined by 271
operator os_enum_type&()
os_type, defined by 348
operator os_field_member_variable&()
os_member, defined by 157
os_member_variable, defined by 168
operator os_function_type&()
os_type, defined by 348
operator os_instantiated_class_type&()
os_class_type, defined by 66
os_type, defined by 348
operator os_integral_type&()
os_type, defined by 348
operator os_literal_template_actual_arg&()
os_template_actual_arg, defined by 323
operator os_member_function&()
os_member, defined by 157
operator os_member_type&()
os_member, defined by 157
operator os_member_variable&()
os_member, defined by 157
operator os_named_indirect_type&()
os_type, defined by 349
operator os_pointer_to_member_type&()
os_type, defined by 349
operator os_pointer_type&()
os_type, defined by 349
operator os_real_type&()
os_type, defined by 349
operator os_reference_type&()
os_type, defined by 349
operator os_relationship_member_ variable&()
os_member, defined by 157
os_member_variable, defined by 168
operator os_template_type_formal&()
os_template_formal_arg, defined by 325
operator os_template_value_formal&()
os_template_formal_arg, defined by 326
operator os_type_template&()
os_template, defined by 322
operator os_type_template_actual_arg&()
os_template_actual_arg, defined by 323
operator os_type_type&()
os_type, defined by 349
operator os_void_type&()
os_type, defined by 349
operator T*()
os_Reference, defined by 206
os_Reference_protected, defined by 228
os_Reference_this_DB, defined by 249
os_Reference_transient, defined by 259
operator void*()
os_reference, defined by 212
os_reference_protected, defined by 234
os_reference_this_DB, defined by 254
os_reference_transient, defined by 264
os_typed_pointer_void, defined by 357
os_access_modifier, the class 44
create() 44
get_base_member() 44
set_base_member() 44
os_access_modifier_cast_to_os_ member() 444

os_access_modifier_create() 445
os_access_modifier_get_base_member() 445
os_access_modifier_set_base_member() 445
~os_address_space_marker()
  os_address_space_marker, defined by 48
os_address_space_marker()
  os_address_space_marker, defined by 47
os_address_space_marker, the class 45–48
  ~os_address_space_marker() 48
  get_current() 46
  get_level() 46
  get_next() 46
  get_previous() 46
  of() 47
  os_address_space_marker() 47
  release() 47
  retain() 47
os_anon_indir_type_cast_to_os_indirect_type() 445
os_anon_indir_type_cast_to_os_type() 445
os_anon_indir_type_create() 445
os_anon_indir_type_get_target_type() 445
os_anon_indir_type_is_const() 446
os_anon_indir_type_is_volatile() 446
os_anon_indir_type_set_is_const() 446
os_anon_indir_type_set_is_volatile() 446
os_anonymous_indirect_type, the class 49
  create() 49
  get_target_type() 49
  is_const() 49
  is_volatile() 49
  set_is_const() 49
  set_is_volatile() 49
os_app_schema, the class 50
  get() 50
os_app_schema_cast_to_os_schema() 446
os_app_schema_get() 446
os_app_schema_get_default() 447
os_array_type, the class 52–53
  create() 52
  get_element_type() 52
  get_number_of_elements() 52
  set_element_type() 52
  set_number_of_elements() 53
os_array_type_cast_to_os_type() 447
os_array_type_create() 447
os_array_type_get_element_type() 447
os_array_type_number_of_elements() 447
os_array_type_set_element_type() 447
os_array_type_set_number_of_elements() 448
os_base_class, the class 54–56
  create() 54
  get_access() 54
  get_class() 54
  get_offset() 55
  get_size() 55
  get_virtual_base_class_pointer_offset() 55
  is_virtual() 55
  Private 55
  Protected 55
  Public 55
  set_access() 55
  set_class() 56
  set_offset() 56
  set_virtual_base_class_no_pointer() 56
  set_virtual_base_class_pointer_offset() 56
  set_virtuals_redefined() 56
  virtual_base_class_has_pointer() 56
  virtuals_redefined() 56
os_base_class_create() 448
os_base_class_get_access() 448
os_base_class_get_class() 448
os_base_class_get_offset() 448
os_base_class_get_offset_in() 448
os_base_class_get_size() 449

os_base_class_get_virtual_base_class_pointer_offset() 449
os_base_class_is_virtual() 449
os_base_class_set_access() 449
os_base_class_set_class() 449
os_base_class_set_is_virtual() 449
os_base_class_set_offset() 450
os_base_class_set_virtual_base_class_no_pointer() 450
os_base_class_set_virtual_base_class_pointer_offset() 450
os_base_class_set_virtuals_redefined() 450
os_base_class_virtual_base_class_has_pointer() 450
os_base_class_virtuals_redefined() 450
OS_BEGIN_TXN(), the macro 378
OS_BEGIN_TXN_NAMED(), the macro 378
os_boolean 7
OS_CACHE_FILE environment variable
  and **objectstore::initialize** 34
os_class_type, the class 57–69
  Anonymous_union 57
  Class 57
  create() 57
  declares_get_os_typespec_function() 58
  defines_virtual_functions() 58
  find_base_class() 58
  find_member_variable() 59
  get_access_of_get_os_typespec_function() 59
  get_allocated_virtual_base_classes() 59
  get_base_classes() 60
  get_class_kind() 60
  get_dispatch_table_pointer_offset() 60
  get_indirect_virtual_base_classes() 61
  get_members() 61
  get_most_derived_class() 61
  get_name() 64
  get_pragmas() 64
  get_size_as_base() 64
  get_source_position() 64
  has_constructor() 65
  has_destructor() 65
  has_dispatch_table() 65
  introduces_virtual_functions() 65
  is_abstract() 65
  is_forward_definition() 65
  is_persistent() 65
  is_template_class() 66
  operator os_instantiated_class_type&() 66
  set_access_of_get_os_typespec_function() 66
  set_base_classes() 66
  set_class_kind() 66
  set_declares_get_os_typespec_function() 67
  set_defines_virtual_functions() 67
  set_dispatch_table_pointer_offset() 67
  set_has_constructor() 67
  set_has_destructor() 67
  set_indirect_virtual_base_classes() 67
  set_introduces_virtual_functions() 67
  set_is_abstract() 67
  set_is_forward_definition() 68
  set_is_persistent() 68
  set_members() 68
  set_name() 68
  set_pragmas() 68
  set_source_position() 68
  Union 69
os_class_type_cast_to_os_instantiated_class_type() 451
os_class_type_cast_to_os_type() 451
os_class_type_create() 451
os_class_type_create_forward_definition() 451
os_class_type_declares_get_os_typespec_function() 451
os_class_type_defines_virtual_functions() 452
os_class_type_find_base_class() 452

os_class_type_find_member() 452
os_class_type_get_access_of_get_os_ typespec_function() 452
os_class_type_get_allocated_virtual_base_ classes() 452
os_class_type_get_base_classes() 452
os_class_type_get_class_kind() 453
os_class_type_get_dispatch_table_pointer_ offset() 453
os_class_type_get_indirect_virtual_base_ classes() 453
os_class_type_get_members() 453
os_class_type_get_most_derived_ class() 453
os_class_type_get_name() 453
os_class_type_get_pragmas() 454
os_class_type_get_size_as_base() 454
os_class_type_get_source_position() 454
os_class_type_has_constructor() 454
os_class_type_has_destructor() 454
os_class_type_has_dispatch_table() 454
os_class_type_introduces_virtual_ functions() 454
os_class_type_is_abstract() 455
os_class_type_is_forward_definition() 455
os_class_type_is_persistent() 455
os_class_type_is_template_class() 455
os_class_type_set_access_of_get_os_ typespec_function() 455
os_class_type_set_base_classes() 455
os_class_type_set_class_kind() 456
os_class_type_set_declares_get_os_ typespec_function() 456
os_class_type_set_defines_virtual_ functions() 456
os_class_type_set_dispatch_table_pointer_ offset() 456
os_class_type_set_has_constructor() 456
os_class_type_set_has_destructor() 456
os_class_type_set_indirect_virtual_base_ classes() 457
os_class_type_set_introduces_virtual_ functions() 457
os_class_type_set_is_abstract() 457
os_class_type_set_is_forward_ definition() 457
os_class_type_set_is_persistent() 457
os_class_type_set_members() 457
os_class_type_set_name() 458
os_class_type_set_pragmas() 458
os_class_type_set_source_position() 458
os_class_typeStruct 68
os_comp_schema, the class 70
  get() 70
os_comp_schema_cast_to_os_ schema() 458
os_comp_schema_get() 458
os_database, the class 71–100
  allow_external_pointers() 72, 203, 209, 215, 220, 224, 231, 237, 242, 367
  change_database_reference() 73
  change_schema_key() 73
  close() 74
  create() 74
  create_root() 76
  create_segment() 76
  decache() 76
  destroy() 77
  find_root() 78
  freeze_schema_key() 78
  get_all_databases() 79
  get_all_roots() 80
  get_all_segments() 80
  get_all_segments_and_permissions() 80
  get_application_info() 81
  get_check_illegal_pointers() 81
  get_database_references() 81
  get_default_check_illegal_pointers() 81
  get_default_lock_whole_segment() 81
  get_default_null_illegal_pointers() 82
  get_default_segment() 82
  get_default_segment_size() 82

get_dirman_host_name() 82
get_fetch_policy() 83
get_file_host_name() 83
get_host_name() 83
get_id() 83
get_incremental_schema_ installation() 83
get_lock_whole_segment() 83
get_n_databases() 84
get_n_roots() 80, 84
get_n_segments() 80, 84
get_opt_cache_lock_mode() 84
get_pathname() 84
get_prms_are_in_standard_format() 85
get_readlock_timeout() 85
get_relative_directory() 85
get_required _DLL_identifiers() 85
get_schema_database() 86
get_sector_size() 86
get_segment() 86
get_transient_database() 86
get_writelock_timeout() 86
insert_required_DLL_identifier() 86
insert_required_DLL_identifiers() 87
is_open() 87
is_open_mvcc() 87
is_open_read_only() 87
is_writable() 87
lookup() 88
of() 88
open() 88
open_mvcc() 89
remove_required_DLL_identifier() 91
set_access_hooks() 91
set_application_info() 93
set_check_illegal_pointers() 93
set_default_check_illegal_pointers() 94
set_default_lock_whole_segment() 94
set_default_null_illegal_pointers() 94
set_default_segment() 94
set_default_segment_size() 95
set_fetch_policy() 95
set_incremental_schema_ installation() 96
set_lock_whole_segment() 97
set_new_id() 97
set_null_illegal_pointers() 97
set_opt_cache_lock_mode() 98
set_readlock_timeout() 98
set_relative_directory() 98
set_schema_database() 99
set_writelock_timeout() 99
size() 99
size_in_sectors() 100
time_created() 100
os_database_alloc() 397, 410
os_database_allow_external_pointers() 410
os_database_change_database_ reference() 411
os_database_change_schema_key() 411
os_database_close() 411
os_database_create() 411
os_database_create_root() 411
os_database_create_segment() 411
os_database_destroy() 412
os_database_find_root() 412
os_database_freeze_schema_key() 412
os_database_get_all_databases() 412
os_database_get_all_roots() 412
os_database_get_all_segments() 412
os_database_get_all_segments_and_ permissions() 413
os_database_get_application_info() 413
os_database_get_database_ references() 413
os_database_get_default_check_illegal_ pointers() 413
os_database_get_default_lock_whole_ segment() 413
os_database_get_default_null_illegal_ pointers() 413

os_database_get_default_read_whole_
segment() 414
os_database_get_default_segment() 414
os_database_get_default_segment_
size() 414
os_database_get_fetch_policy() 414
os_database_get_host_name() 414
os_database_get_id() 414
os_database_get_incremental_schema_
installation() 414
os_database_get_lock_whole_
segment() 415
os_database_get_n_databases() 415
os_database_get_n_roots() 415
os_database_get_n_segments() 415
os_database_get_null_illegal_pointers() 415
os_database_get_opt_cache_lock_
mode() 415
os_database_get_pathname() 415
os_database_get_readlock_timeout() 415
os_database_get_relative_directory() 416
os_database_get_sector_size() 416
os_database_get_segment() 416
os_database_get_transient_database() 416
os_database_get_writelock_timeout() 416
os_database_is_open() 416
os_database_is_open_mvcc() 416
os_database_is_open_read_only 417
os_database_is_writable() 417
os_database_lookup() 417
os_database_lookup_open() 417
os_database_lookup_open_mvcc() 417
os_database_of() 417
os_database_open() 417
os_database_open_mvcc() 418
~os_database_reference()
os_database_reference, defined by 101
os_database_reference()
os_database_reference, defined by 101
os_database_reference, the class 101–102
delete_array() 101
get_name() 101
operator ==() 101
~os_database_reference() 101
os_database_reference() 101
os_database_reference_delete_array() 422
os_database_reference_equal() 422
os_database_reference_free() 422
os_database_reference_get_name() 422
~os_database_root()
os_database_root, defined by 103
os_database_root, the class 103–104
find() 103
get_name() 103
get_typespec() 103
get_value() 103
~os_database_root() 103
set_value() 104
os_database_root_find() 423
os_database_root_free() 423
os_database_root_get_name() 423
os_database_root_get_typespec() 423
os_database_root_get_value() 423
os_database_root_set_value() 423
os_database_schema, the class 105
get() 105
get_for_update() 105
install() 105
os_database_schema_cast_to_os_
schema() 458
os_database_schema_get() 459
os_database_schema_get_for_update() 459
os_database_schema_install() 459
os_database_schema_install_with_
options() 459
os_database_set_access_hooks() 418
os_database_set_application_info() 418
os_database_set_check_illegal_
pointers() 418
os_database_set_default_check_illegal_
pointers() 418

os_database_set_default_lock_whole_segment() 418
os_database_set_default_null_illegal_pointers() 419
os_database_set_default_segment() 419
os_database_set_default_segment_size() 419
os_database_set_fetch_policy() 419
os_database_set_incremental_schema_installation() 419
os_database_set_lock_whole_segment() 419
os_database_set_new_id() 419
os_database_set_null_illegal_pointers() 420
os_database_set_opt_cache_lock_mode() 420
os_database_set_readlock_timeout() 420
os_database_set_relative_directory() 420
os_database_set_writelock_timeout() 420
os_database_size() 420
os_database_size_in_sectors() 420
os_database_time_created() 421
os_dbutil, the class 106–127
- **chgrp()** 106
- **chmod()** 106
- **chown()** 106
- **close_all_server_connections()** 107
- **close_server_connection()** 107
- **cmgr_remove_file()** 107
- **cmgr_shutdown()** 107
- **cmgr_stat()** 108
- **compare_schemas()** 110
- **copy_database()** 110
- **disk_free()** 110
- **expand_global()** 111
- **get_client_name()** 111
- **get_sector_size()** 112
- **initialize()** 112
- **list_directory()** 112
- **make_link()** 112
- **mkdir()** 113
- **osprmgc()** 117
- **ossize()** 113
- **osverifydb()** 114
- **osverifydb_one_segment()** 116
- **rehost_all_links()** 117
- **rehost_link()** 117
- **remove()** 118
- **rename()** 118
- **rmdir()** 118
- **set_application_schema_path()** 119
- **set_client_name()** 119
- **stat()** 119
- **svr_checkpoint()** 119
- **svr_client_kill()** 120
- **svr_ping()** 120
- **svr_shutdown()** 121
- **svr_stat()** 121

os_dbutil_chgrp() 424
os_dbutil_chmod() 424
os_dbutil_chown() 424
os_dbutil_close_all_connections() 424
os_dbutil_close_all_server_connections() 424
os_dbutil_close_server_connection() 424
os_dbutil_cmgr_log() 425
os_dbutil_cmgr_remove_file() 425
os_dbutil_cmgr_shutdown() 425
os_dbutil_cmgr_stat() 425
os_dbutil_compare_schemas() 425
os_dbutil_copy_database() 425
os_dbutil_delete_os_rawfs_entries() 426
os_dbutil_disk_free() 426
os_dbutil_expand_global() 426
os_dbutil_get_client_name() 426
os_dbutil_get_sector_size() 426
os_dbutil_initialize() 426
os_dbutil_list_directory() 426
os_dbutil_make_link() 427
os_dbutil_mkdir() 427
os_dbutil_ossize() 427
os_dbutil_osverifydb() 427

os_dbutil_osverifydb_one_segment() 427
os_dbutil_rehost_all_links() 428
os_dbutil_rehost_link() 428
os_dbutil_remove() 428
os_dbutil_rename() 428
os_dbutil_rmdir() 428
os_dbutil_set_application_schema_ path() 428
os_dbutil_set_client_name() 429
os_dbutil_stat() 429
os_dbutil_svr_checkpoint() 429
os_dbutil_svr_client_kill() 429
os_dbutil_svr_ping() 429
os_dbutil_svr_shutdown() 429
os_dbutil_svr_stat() 430
OS_DEF_EXCEPT_ACTION environment variable 559, 560
os_delete_tix_exception() 552
os_DLL_finder, the class 128–129
  equal_DLL_identifiers() 128
  get() 128
  register_() 128
  unregister() 128
os_DLL_schema_info, the class
  add_DLL_identifier() 130
  DLL_loaded() 130, 131
  DLL_unloaded() 131
~os_dynamic_extent()
  os_dynamic_extent, defined by 134
os_dynamic_extent()
  os_dynamic_extent, defined by 132
os_dynamic_extent, the class 132–134
  ~os_dynamic_extent() 134
  insert() 133
  os_dynamic_extent() 132
  remove() 133
OS_END_FAULT_HANDLER, the macro 383
OS_END_TXN(), the macro 378
os_enum_type, the class 135–136
  create() 135
  get_enumerator() 135
  get_enumerators() 135
  get_name() 135
  get_pragmas() 136
  get_source_position() 136
  set_enumerators() 136
  set_name() 136
  set_pragmas() 136
  set_source_position() 136
os_enum_type_cast_to_os_type() 459
os_enum_type_create() 459
os_enum_type_get_enumerators() 460
os_enum_type_get_name() 460
os_enum_type_get_pragmas() 460
os_enum_type_get_source_position() 460
os_enum_type_set_enumerators() 460
os_enum_type_set_name() 460
os_enum_type_set_pragmas() 461
os_enum_type_set_source_position() 461
os_enumerator_literal, the class 137
  create() 137
  get_name() 137
  get_value() 137
  set_name() 137
  set_value() 137
os_enumerator_literal_create() 461
os_enumerator_literal_get_name() 461
os_enumerator_literal_get_value() 461
os_enumerator_literal_set_name() 461
os_enumerator_literal_set_value() 462
OS_ESTABLISH_FAULT_HANDLER, the macro 383
os_evolve_subtype_fun_binding()
  os_evolve_subtype_fun_binding, defined by 138
os_evolve_subtype_fun_binding, the class 138
  os_evolve_subtype_fun_binding() 138
os_evolve_subtype_fun_binding_ create() 543
os_evolve_subtype_fun_binding_ destroy() 543

os_failover_server, the class 139–140
  get_logical_server_hostname() 139
  get_online_server_hostname() 139
  get_reconnect_retry_interval() 139
  get_reconnect_timeout() 139
  set_reconnect_timeout_and_ interval() 140
os_fetch, the global function 370
os_fetch_address, the global function 373
os_fetch_address_os_base_class() 439
os_fetch_address_os_member_ variable() 439
os_fetch_page fetch policy 95, 302
os_fetch_segment fetch policy 95, 302
os_fetch_stream fetch policy 95, 302
os_field_member_variable, the class 141
  create() 141
  get_offset() 141
  get_size() 141
  set_size() 141
os_field_member_variable_create() 462
os_field_member_variable_get_offset() 462
os_field_member_variable_get_size() 462
os_field_member_variable_set_offset() 462
os_field_member_variable_set_size() 463
os_fld_mem_var_cast_to_os_mem_ variable() 463
os_fld_mem_var_cast_to_os_member() 463
os_function_type, the class 142–143
  create() 142
  equal_signature() 142
  get_arg_list() 142
  get_arg_list_kind() 143
  get_return_type() 143
  Known 143
  set_arg_list() 143
  set_arg_list_kind() 143
  set_return_type() 143
  Unknown 143
  Variable 143
os_function_type_cast_to_os_type() 463
os_function_type_create() 463
os_function_type_equal_signature() 463
os_function_type_get_arg_list() 464
os_function_type_get_arg_list_kind() 464
os_function_type_get_return_type() 464
os_function_type_set_arg_list() 464
os_function_type_set_arg_list_kind() 464
os_function_type_set_return_type() 464
os_indir_type_cast_to_os_anon_indir_ type() 465
os_indir_type_cast_to_os_named_indir_ type() 465
os_indir_type_get_target_type() 465
os_indir_type_set_target_type() 465
os_indirect_type, the class 144
  get_target_type() 144
  set_target_type() 144
os_indirect_type_cast_to_os_type() 465
os_inst_class_type_cast_to_os_class_ type() 465
os_inst_class_type_cast_to_os_type() 466
os_instantiated_class_type, the class 145
  create() 145
  get_instantiation() 145
  set_instantiation() 145
os_instantiated_class_type_create() 466
os_instantiated_class_type_create_forward_ definition() 466
os_instantiated_class_type_get_ instantiation() 466
os_instantiated_class_type_set_ instantiation() 466
os_int32 7
os_integral_type, the class 146–147
  create() 146
  create_defaulted_char() 146
  create_int() 146
  create_long() 146
  create_short() 146
  create_signed_char() 147
  create_unsigned_char() 147

is_signed() 147
os_integral_type_cast_to_os_type() 466
os_integral_type_create() 467
os_integral_type_create_defaulted_char() 467
os_integral_type_create_int() 467
os_integral_type_create_long() 467
os_integral_type_create_short() 467
os_integral_type_create_signed_char() 467
os_integral_type_create_unsigned_char() 467
os_integral_type_create_wchar_t() 468
os_integral_type_is_signed() 468
os_keyword_arg_char_create() 433
os_keyword_arg_double_create() 435
os_keyword_arg_float_create() 435
os_keyword_arg_int_create() 434
os_keyword_arg_list_create() 436
os_keyword_arg_list_delete() 436
os_keyword_arg_long_create() 434
os_keyword_arg_long_double_create() 435
os_keyword_arg_pointer_create() 435
os_keyword_arg_short_create() 433
os_keyword_arg_uchar_create() 433
os_keyword_arg_uint_create() 434
os_keyword_arg_ulong_create() 434
os_keyword_arg_ushort_create() 433
os_literal, the class 148–150
  create_char() 148
  create_enum_literal() 148
  create_pointer_literal() 148
  create_signed_char() 148
  create_signed_int() 148
  create_signed_long() 148
  create_signed_short() 148
  create_unsigned_char() 149
  create_unsigned_int() 149
  create_unsigned_long() 149
  create_unsigned_short() 149
  create_wchar_t() 149
  get_char_value() 149
  get_enum_literal() 149
  get_kind() 149
  get_pointer_literal() 150
  get_signed_integral_value() 150
  get_type() 150
  get_unsigned_integral_value() 150
  get_wchar_t_value() 150
os_literal_create_char() 468
os_literal_create_enum_literal() 468
os_literal_create_pointer_literal() 468
os_literal_create_signed_char() 468
os_literal_create_signed_int() 468, 469
os_literal_create_signed_long() 469
os_literal_create_unsigned_char() 469
os_literal_create_unsigned_int() 469
os_literal_create_unsigned_long() 469
os_literal_create_unsigned_short() 469
os_literal_create_wchar_t() 470
os_literal_get_char_value() 470
os_literal_get_enum_literal() 470
os_literal_get_kind() 470
os_literal_get_pointer_literal() 470
os_literal_get_signed_integral_value() 470
os_literal_get_type() 470
os_literal_get_unsigned_integral_value() 471
os_literal_get_wchar_t_value() 471
os_literal_is_unspecified() 471
os_literal_template_actual_arg() 471
os_literal_template_actual_arg, the class 151
  create() 151
  get_literal() 151
  set_literal() 151
os_literal_template_actual_arg_get_literal() 471
os_literal_template_actual_arg_set_literal() 471
os_lock_timeout_exception, the class 152–153
  get_application_names() 152, 565

get_fault_addr() 152, 565
get_hostnames() 152, 565
get_lock_type() 153, 565
get_pids() 153, 565
number_of_blockers() 153, 565
os_lock_timeout_exception_get_application_names() 437
os_lock_timeout_exception_get_fault_addr() 437
os_lock_timeout_exception_get_hostnames() 437
os_lock_timeout_exception_get_lock_type() 437
os_lock_timeout_exception_get_pids() 437
os_lock_timeout_exception_number_of_blockers() 438
OS_LOG_FILE environment variable
and **objectstore::set_log_file()** 38
OS_MARK_QUERY_FUNCTION(), the macro 385
OS_MARK_SCHEMA_TYPESPEC(), the macro 386
os_mem_function_cast_to_os_member() 472
os_mem_var_cast_to_os_field_member_variable() 472
os_mem_var_cast_to_os_member() 472
os_mem_var_cast_to_os_relationship_member_variable() 472
os_member, the class 154–158
Access_modifier 154
defining_class() 154
Field_variable 154
Function 154
get_access() 154
get_defining_class() 155
get_kind() 155
is_unspecified() 155
Namespace 155
operator const os_access_modifier&() 155
operator const os_field_member_variable&() 156
operator const os_member_function&() 156
operator const os_member_type&() 156
operator const os_member_variable&() 156
operator const os_relationship_member_variable&() 156
operator os_access_modifier&() 156
operator os_field_member_variable&() 157
operator os_member_function&() 157
operator os_member_type&() 157
operator os_member_variable&() 157
operator os_relationship_member_variable&() 157
Private 157
Protected 157
Public 158
Relationship 158
set_access() 158
Type 158
Variable 158
os_member_cast_to_os_access_modifier() 472
os_member_cast_to_os_field_member_variable() 472
os_member_cast_to_os_member_function() 473
os_member_cast_to_os_member_type() 473
os_member_cast_to_os_member_variable() 473
os_member_cast_to_os_relationship_member_variable() 473
os_member_defining_class() 473
os_member_function, the class 159–163
create() 159
get_call_linkage() 159
get_function_kind() 159

get_name() 160
get_source_position() 160
get_type() 160
is_const() 160
is_inline() 160
is_overloaded() 161
is_pure_virtual() 161
is_static() 161
is_virtual() 161
is_volatile() 161
set_call_linkage() 161
set_is_const() 161
set_is_inline() 162
set_is_overloaded() 162
set_is_pure_virtual() 162
set_is_static() 162
set_is_virtual() 162
set_is_volatile() 162
set_name() 162
set_source_position() 162
set_type() 163
os_member_function_create() 473
os_member_function_get_call_linkage() 474
os_member_function_get_function_kind() 474
os_member_function_get_name() 474
os_member_function_get_source_position() 474
os_member_function_get_type() 474
os_member_function_is_const() 474
os_member_function_is_inline() 475
os_member_function_is_overloaded() 475
os_member_function_is_pure_virtual() 475
os_member_function_is_static() 475
os_member_function_is_virtual() 475
os_member_function_is_volatile() 475
os_member_function_set_call_linkage() 476
os_member_function_set_function_kind() 476
os_member_function_set_is_const() 476
os_member_function_set_is_inline() 476
os_member_function_set_is_overloaded() 476
os_member_function_set_is_pure_virtual() 476
os_member_function_set_is_static() 477
os_member_function_set_is_virtual() 477
os_member_function_set_is_volatile() 477
os_member_function_set_name() 477
os_member_function_set_source_position() 477
os_member_function_set_type() 477
os_member_get_access() 478
os_member_get_kind() 478
os_member_get_storage_class() 478
os_member_is_unspecified() 478
os_member_namespace, the class 164
create() 164
get_namespace() 164
set_namespace() 164
os_member_set_access() 478
os_member_type, the class 165
create() 165
get_type() 165
set_type() 165
os_member_type_cast_to_os_member() 478
os_member_type_create() 479
os_member_type_get_type() 479
os_member_type_set_type() 479
os_member_variable, the class 166–169
create() 166
get_name() 166
get_offset() 167
get_size() 167
get_source_position() 167
get_storage_class() 167
get_type() 166
is_field() 167
is_persistent() 168
is_static() 167

operator const os_field_member_
variable&() 168
operator const os_relationship_member_
variable&() 168
operator os_field_member_
variable&() 168
operator os_relationship_member_
variable&() 168
Persistent 167
Regular 167
set_name() 168
set_offset() 169
set_source_position() 169
set_storage_class() 169
set_type() 169
Static 167
os_member_variable_create() 479
os_member_variable_get_name() 479
os_member_variable_get_offset() 479
os_member_variable_get_size() 480
os_member_variable_get_source_
position() 480
os_member_variable_get_type() 480
os_member_variable_is_field() 480
os_member_variable_is_persistent() 480
os_member_variable_is_static() 480
os_member_variable_set_name() 481
os_member_variable_set_offset() 481
os_member_variable_set_source_
position() 481
os_member_variable_set_storage_
class() 481
os_member_variable_set_type() 481
os_mop, the class 170–173
bind() 170
copy_classes() 171
current() 171
find_namespace() 171
find_type() 172
get_failure_classes() 172
get_neutralized_classes() 172
get_transient_schema() 172
initialize() 172
initialize_object_metadata() 172
reset() 173
os_mop_copy_classes() 481
os_mop_find_type() 482
os_mop_get_transient_schema() 482
os_mop_initialize() 482
os_mop_initialize_object_metadata() 482
os_named_indir_type_cast_to_os_indir_
type() 482
os_named_indir_type_cast_to_os_
type() 482
os_named_indir_type_get_target_type() 482
os_named_indirect_type, the class 174
create() 174
get_name() 174
get_source_position() 174
set_name() 174
set_source_position() 174
os_named_indirect_type_create() 483
os_named_indirect_type_get_name() 483
os_named_indirect_type_get_source_
position() 483
os_named_indirect_type_set_name() 483
os_named_indirect_type_set_source_
position() 483
os_namespace, the class 175
create() 175
get_enclosing_namespace() 175
get_members() 175
get_name() 175
set_enclosing_namespace() 175
set_members() 175
set_name() 175
os_new_database_reference_name() 422
os_new_tix_exception() 552
os_notification()
os_notification, defined by 176
os_notification, the class 176–185
_get_fd() 177

assign() 176
get_database() 176
get_kind() 177
get_reference() 177
get_string() 177
notify_immediate() 178
notify_on_commit() 179
os_notification() 176
queue_status() 180
receive() 181
set_queue_size() 182
subscribe() 182
unsubscribe() 183
os_notification__get_fd() 502
os_notification_assign() 500
os_notification_free() 500
os_notification_free_array() 500
os_notification_get_database() 501
os_notification_get_kind() 501
os_notification_get_reference() 501
os_notification_get_string() 501
os_notification_new() 500
os_notification_new_array() 500
os_notification_notify_immediate() 501
os_notification_notify_on_commit() 502
os_notification_queue_status() 502
os_notification_receive() 502
os_notification_set_queue_size() 502
os_notification_sizeof() 500
os_notification_subscribe() 501
os_notification_unsubscribe() 501
os_object_cluster, the class 186–188, 296
destroy() 186
destroy_always 187
destroy_if_empty 187
get_info() 187
is_empty() 187
of() 187
segment_of() 187
os_object_cluster_destroy() 505
os_object_cluster_free() 505
os_object_cluster_get_info() 505
os_object_cluster_is_empty() 505
os_object_cluster_of() 505
os_object_cluster_segment_of() 506
~os_object_cursor()
os_object_cursor, defined by 190
os_object_cursor()
os_object_cursor, defined by 190
os_object_cursor, the class 189–190
current() 189
first() 189
more() 190
next() 190
~os_object_cursor() 190
os_object_cursor() 190
set() 190
os_object_cursor_create() 507
os_object_cursor_current() 507
os_object_cursor_delete() 507
os_object_cursor_first() 507
os_object_cursor_more() 507
os_object_cursor_next() 508
os_object_cursor_set() 508
os_pathname_and_segment_number()
os_pathname_and_segment_number, defined by 191
os_pathname_and_segment_number, the class 191
database_pathname 191
os_pathname_and_segment_number() 191
segment_number 191
os_pointer_literal, the class 192
create() 192
get_name() 192
get_type() 192
set_name() 192
set_type() 192
os_pointer_literal_create() 483
os_pointer_literal_get_name() 484
os_pointer_literal_get_type() 484

os_pointer_literal_set_name() 484
os_pointer_literal_set_type() 484
os_pointer_to_member_type, the class 193
  create() 193
  get_target_class() 193
  set_target_class() 193
os_pointer_to_member_type_create() 484
os_pointer_to_member_type_get_target_class() 484
os_pointer_to_member_type_set_target_class() 485
os_pointer_type, the class 194
  create() 194
  get_target_type() 194
  set_target_type() 194
os_pointer_type_cast_to_os_reference_type() 485
os_pointer_type_cast_to_os_type() 485
os_pointer_type_create() 485
os_pointer_type_get_target_type() 485
os_pointer_type_set_target_type() 485
os_pragma, the class 195
  create() 195
  get_string() 195
  is_recognized() 195
os_pragma_create_pragma() 486
os_pragma_get_string() 486
os_pragma_is_recognized() 486
os_prim_typed_pointer_void_assign() 551
os_prim_typed_pointer_void_get_pointer() 543
os_prim_typed_pointer_void_get_type() 543
os_ptr_to_mem_type_cast_to_os_ptr_type() 486
os_ptr_to_mem_type_cast_to_os_type() 486
os_ptr_type_cast_to_os_ptr_to_mem_type() 486
~os_pvar()
  os_pvar, defined by 198
os_pvar()
  os_pvar, defined by 196, 197
os_pvar, the class 196–198
  init_int() 197, 198
  init_long() 197, 198
  init_pointer() 197, 198
  ~os_pvar() 198
  os_pvar() 196, 197
os_pvar_define() 509
os_pvar_init_int() 509
os_pvar_init_long() 509
os_pvar_init_pointer() 509
os_pvar_undefine() 509
~os_rawfs_entry()
  os_rawfs_entry, defined by 201
os_rawfs_entry()
  os_rawfs_entry, defined by 201
os_rawfs_entry, the class 199–201
  get_abs_path() 199
  get_creation_time() 199
  get_group_name() 199
  get_link_host() 199
  get_link_path() 199
  get_n_sectors() 199
  get_name() 199
  get_permission() 200
  get_server_host() 200
  get_type() 200
  get_user_name() 200
  is_db() 200
  is_dir() 200
  is_link() 200
  operator =() 201
  ~os_rawfs_entry() 201
  os_rawfs_entry() 201
  OSU_DATABASE 201
  OSU_DIRECTORY 201
  OSU_LINK 201
os_rawfs_entry_delete() 510
os_rawfs_entry_get_creation_time() 510
os_rawfs_entry_get_group_name() 510

os_rawfs_entry_get_link_host() 510
os_rawfs_entry_get_link_path() 510
os_rawfs_entry_get_n_sectors() 511
os_rawfs_entry_get_name() 511
os_rawfs_entry_get_permission() 511
os_rawfs_entry_get_server_host() 511
os_rawfs_entry_get_type() 511
os_rawfs_entry_get_user_name() 511
os_rawfs_entry_is_db() 511
os_rawfs_entry_is_dir() 512
os_rawfs_entry_is_link() 512
os_real_type, the class 202
  create() 202
  create_double() 202
  create_float() 202
  create_long_double() 202
os_real_type_cast_to_os_type() 486
os_real_type_create() 487
os_real_type_create_double() 487
os_real_type_create_float() 487
os_real_type_create_long_double() 487
os_ref_type_cast_to_os_ptr_type() 487
os_ref_type_cast_to_os_type() 487
os_Reference()
  os_Reference, defined by 208
os_reference()
  os_reference, defined by 213
os_Reference, the class 203–208
  dump() 205
  get_database() 205
  get_database_key() 205
  get_open_database() 205
  get_os_typespec() 206
  hash() 206
  load() 206
  operator !() 207
  operator !=() 207
  operator <() 207
  operator <=() 208
  operator =() 206
  operator ==() 207
  operator >() 207
  operator –>() 206
  operator >=() 208
  operator T*() 206
  os_Reference() 208
  resolve() 208
os_reference, the class 209–214
  dump() 210
  get_database() 211
  get_database_key() 211
  get_open_database() 211
  get_os_typespec() 211
  hash() 211
  load() 211
  operator !=() 212
  operator <() 213
  operator <=() 213
  operator =() 212
  operator ==() 212
  operator >() 213
  operator >=() 213
  operator void*() 212
  os_reference() 213
  resolve() 214
os_reference_dump() 530
os_reference_dump2() 530
os_reference_EQ() 529
os_reference_GE() 529
os_reference_get_database_key() 530
os_reference_GT() 529
os_reference_LE() 529
os_reference_load2() 530
os_Reference_local()
  os_Reference_local, defined by 218
os_reference_local()
  os_reference_local, defined by 223
os_Reference_local, the class 215–219
  dump() 216
  get_database_key() 216
  get_os_typespec() 217
  hash() 217

load() 217
operator !=() 217
operator <() 218
operator <=() 218
operator =() 217
operator ==() 217
operator >() 218
operator >=() 218
os_Reference_local() 218
resolve() 218
os_reference_local, the class 220–223, 275
dump() 221
get_database_key() 221
get_os_typespec() 221
hash() 221
load() 221
operator !=() 222
operator <() 222
operator <=() 223
operator =() 221
operator ==() 222
operator >() 222
operator >=() 222
os_reference_local() 223
resolve() 223
os_reference_local_dump() 532
os_reference_local_EQ() 531
os_reference_local_GE() 531
os_reference_local_get_database_key() 532
os_reference_local_GT() 531
os_reference_local_LE() 532
os_reference_local_LT() 532
os_reference_local_NE() 532
os_reference_local_new() 533
os_reference_local_new_from_string() 533
os_reference_local_NOT() 532
os_reference_local_resolve() 533
os_reference_LT() 529
os_reference_NE() 530
os_reference_new() 531
os_reference_new_from_string() 531
os_reference_NOT() 530
os_Reference_protected()
os_Reference_protected, defined by 230
os_reference_protected()
os_reference_protected, defined by 236
os_Reference_protected, the class 224–230
deleted() 226
dump() 226
forget() 226
get_database() 227
get_database_key() 227
get_open_database() 227
get_os_typespec() 227
hash() 227
load() 227
operator !() 228
operator !=() 229
operator <() 229
operator <=() 229
operator =() 228
operator ==() 228
operator >() 229
operator –>() 228
operator >=() 229
operator T*() 228
os_Reference_protected() 230
resolve() 230
os_reference_protected, the class 231–236
deleted() 232
dump() 232
forget() 233
get_database() 233
get_database_key() 233
get_open_database() 233
get_os_typespec() 233
hash() 234
load() 234
operator !() 235
operator !=() 235
operator <() 235
operator <=() 235

operator =() 234
operator ==() 234
operator >() 235
operator >=() 235
operator void*() 234
os_reference_protected() 236
resolve() 236
os_reference_protected_deleted() 534
os_reference_protected_dump() 534
os_reference_protected_dump2() 534
os_reference_protected_EQ() 533
os_reference_protected_forget() 535
os_reference_protected_GE() 533
os_reference_protected_get_database_key() 535
os_reference_protected_GT() 533
os_reference_protected_LE() 534
os_reference_protected_load2() 535
os_Reference_protected_local()
os_Reference_protected_local, defined by 241
os_reference_protected_local()
os_reference_protected_local, defined by 246
os_Reference_protected_local, the class 237–241
deleted() 238
dump() 238
forget() 239
get_database_key() 239
get_os_typespec() 239
hash() 239
load() 239
operator !() 240
operator !=() 240
operator <() 240
operator <=() 241
operator =() 239
operator ==() 240
operator >() 240
operator >=() 241
os_Reference_protected_local() 241
resolve() 241
os_reference_protected_local, the class 242–246, 275
deleted() 243
dump() 243
forget() 243
get_database_key() 243
get_os_typespec() 243
hash() 244
load() 244
operator !() 244
operator !=() 244
operator <() 245
operator <=() 245
operator =() 244
operator ==() 244
operator >() 245
operator >=() 245
os_reference_protected_local() 246
resolve() 246
os_reference_protected_local_deleted() 536
os_reference_protected_local_dump() 536
os_reference_protected_local_EQ() 535
os_reference_protected_local_forget() 536
os_reference_protected_local_get_database_key() 537
os_reference_protected_local_GT() 535
os_reference_protected_local_LE() 536
os_reference_protected_local_LT() 536
os_reference_protected_local_NE() 536
os_reference_protected_local_new() 537
os_reference_protected_local_new_from_string() 537
os_reference_protected_local_resolve() 537
os_reference_protected_LT() 534
os_reference_protected_NE() 534
os_reference_protected_new() 537
os_reference_protected_new_from_string() 537
os_reference_protected_resolve() 538

os_reference_resolve() 531
os_Reference_this_DB()
  os_Reference_this_DB, defined by 251
os_reference_this_DB()
  os_reference_this_DB, defined by 255
os_Reference_this_DB, the class 247–251
  dump() 248
  get_database() 248
  get_database_key() 249
  get_os_typespec() 249
  hash() 249
  load() 249
  operator !=() 250
  operator <() 250
  operator <=() 251
  operator =() 250
  operator ==() 250
  operator >() 250
  operator –>() 250
  operator >=() 251
  operator T*() 249
  os_Reference_this_DB() 251
  resolve() 251
os_reference_this_DB, the class 252–256
  dump() 253
  get_database() 253
  get_database_key() 253
  get_os_typespec() 253
  hash() 253
  load() 254
  operator !=() 254
  operator <() 255
  operator <=() 255
  operator =() 254
  operator ==() 254
  operator >() 255
  operator >=() 255
  operator void*() 254
  os_reference_this_DB() 255
  resolve() 256
os_reference_this_DB_dump() 539
os_reference_this_DB_dump2() 539
os_reference_this_DB_EQ() 538
os_reference_this_DB_GE() 538
os_reference_this_DB_get_database_key() 540
os_reference_this_DB_GT() 538
os_reference_this_DB_LE() 538
os_reference_this_DB_LT() 538
os_reference_this_DB_NE() 539
os_reference_this_DB_new() 539
os_reference_this_DB_new_from_string() 539
os_reference_this_DB_NOT() 539
os_reference_this_DB_resolve() 539
os_Reference_transient()
  os_Reference_transient, defined by 261
os_reference_transient()
  os_reference_transient, defined by 265
os_Reference_transient, the class 257–261
  dump() 258
  get_database_key() 258
  hash() 259
  load() 259
  operator !=() 260
  operator <() 260
  operator <=() 261
  operator =() 259
  operator ==() 260
  operator >() 260
  operator –>() 259
  operator >=() 260
  operator T*() 259
  os_Reference_transient() 261
  resolve() 261
os_reference_transient, the class 262–265
  dump() 262
  get_database_key() 263
  hash() 263
  load() 263
  operator !=() 264
  operator <() 264

operator <=() 265
operator =() 264
operator ==() 264
operator >() 264
operator >=() 265
operator void*() 264
os_reference_transient() 265
resolve() 265
os_reference_transient_dump() 541
os_reference_transient_dump2() 542
os_reference_transient_EQ() 540
os_reference_transient_GE() 540
os_reference_transient_get_database_ key() 542
os_reference_transient_GT() 540
os_reference_transient_LE() 540
os_reference_transient_load2() 542
os_reference_transient_LT() 541
os_reference_transient_NE() 541
os_reference_transient_new() 541
os_reference_transient_new_from_ string() 541
os_reference_transient_NOT() 541
os_reference_transient_resolve() 542
os_reference_type, the class 266
create() 266
os_reference_type_create() 487
os_rel_mem_var_cast_to_os_mem_ variable() 488
os_rel_mem_var_cast_to_os_member() 488
os_rel_mem_var_get_related_class() 488
os_rel_mem_var_get_related_member() 488
os_relationship_member_variable, the class 267
get_related_class() 267
get_related_member() 267
OS_REPORT_DLL_LOAD_AND_UNLOAD component schema generation 387
os_retain_address()
os_retain_address, defined by 269
os_retain_address, the class 268–269
os_retain_address() 269
release() 269
retaining() 269
set_retain() 269
os_schema, the class 270–271
Application_schema 270
Compilation_schema 270
Database_schema 270
find_type() 270
get_classes() 270
get_kind() 270
operator const os_app_schema&() 270
operator const os_comp_schema&() 271
operator const os_database_ schema&() 271
operator os_app_schema&() 271
operator os_comp_schema&() 271
operator os_database_schema&() 271
os_schema_cast_to_os_app_schema() 488
os_schema_cast_to_os_comp_ schema() 488
os_schema_cast_to_os_database_ schema() 489
OS_SCHEMA_DLL_ID component schema generation 387
os_schema_evolution, the class 272–288
augment_classes_to_be_recycled() 276
augment_classes_to_be_removed() 277
augment_post_evol_transformers() 277
augment_subtype_selectors() 278
evolve() 278
get_evolved_address() 283
get_evolved_object() 283
get_ignore_illegal_pointers() 283
get_unevolved_address() 283
get_unevolved_object() 284
get_work_database() 284
set_evolved_schema_db_name() 284
set_ignore_illegal_pointers() 284
set_illegal_pointer_handler() 284

set_local_references_are_db_relative() 286
set_obsolete_index_handler() 286
set_obsolete_query_handler() 286
set_task_list_file_name() 287
task_list() 287
os_schema_evolution_augment_classes_to_be_recycled() 543
os_schema_evolution_augment_classes_to_be_recycled_multiple() 544
os_schema_evolution_augment_classes_to_be_removed() 544
os_schema_evolution_augment_classes_to_be_removed_multiple() 544
os_schema_evolution_augment_nonversioned_transformers() 544
os_schema_evolution_augment_nonversioned_transformers_multiple() 544
os_schema_evolution_augment_post_evol_transformers() 545
os_schema_evolution_augment_post_evol_transformers_multiple() 545
os_schema_evolution_augment_pre_evol_transformers() 545
os_schema_evolution_augment_pre_evol_transformers_multiple() 545
os_schema_evolution_augment_subtype_selectors() 545
os_schema_evolution_augment_subtype_selectors_multiple() 545
os_schema_evolution_evolve() 546
os_schema_evolution_evolve_multiple() 546
os_schema_evolution_get_enclosing_object() 546
os_schema_evolution_get_evolved_address() 546
os_schema_evolution_get_evolved_object() 546
os_schema_evolution_get_ignore_illegal_pointers() 546
os_schema_evolution_get_obsolete_index_handler() 547
os_schema_evolution_get_obsolete_query_handler() 547
os_schema_evolution_get_path_to_member() 547
os_schema_evolution_get_unevolved_address() 547
os_schema_evolution_get_unevolved_object() 547
os_schema_evolution_get_work_database() 547
os_schema_evolution_set_evolved_schema_db_name() 548
os_schema_evolution_set_explanation_level() 548
os_schema_evolution_set_ignore_illegal_pointers() 548
os_schema_evolution_set_illegal_pointer_handler_database_root() 548
os_schema_evolution_set_illegal_pointer_handler_pointer() 548
os_schema_evolution_set_illegal_pointer_handler_pointer_to_member() 548
os_schema_evolution_set_illegal_pointer_handler_reference_local() 549
os_schema_evolution_set_illegal_pointer_handler_reference_version() 549
os_schema_evolution_set_local_references_are_db_relative() 549
os_schema_evolution_set_obsolete_index_handler() 549
os_schema_evolution_set_obsolete_query_handler() 550
os_schema_evolution_set_pre_evolution_setup_hook() 550
os_schema_evolution_set_recycle_all_classes() 550

os_schema_evolution_set_resolve_ambiguous_void_pointers() 550
os_schema_evolution_set_task_list_file_name() 550
os_schema_evolution_task_list() 550
os_schema_evolution_task_list_multiple() 551
os_schema_find_type() 489
os_schema_get_classes() 489
os_schema_get_kind() 489
os_schema_handle, the class 289–291
  DLL_unloaded() 289
  get() 289
  get_all() 289
  get_application_schema_handle() 290
  get_DLL_identifiers() 290
  get_schema_database() 290
  get_schema_database_pathname() 290
  get_schema_info() 291
  get_status() 291
  insert_required_DLL_identifiers() 291
  set_schema_database_pathname() 291
os_schema_handle_status, the enumerator 292
os_schema_info, the class 293, 293–??
  get() 293
  get_DLL_identifiers() 293
  get_schema_database_pathname() 293
  set_schema_database_pathname() 293
OS_SCHEMA_INFO_NAME component schema generation 387
os_schema_install_options_create() 489
os_schema_install_options_destroy() 489
os_schema_install_options_get_copy_member_functions() 490
os_schema_install_options_set_copy_member_functions() 490
os_segment, the class 295–306
  allow_external_pointers() 220, 224, 231, 237, 242, 295, 367
  create_object_cluster() 296
  database_of() 296
  destroy() 296
  external_pointer_status() 297
  get_access_control() 297
  get_all_object_clusters() 297
  get_application_info() 297
  get_check_illegal_pointers() 298
  get_comment() 298
  get_database_references() 298
  get_fetch_policy() 298
  get_lock_whole_segment() 298
  get_n_object_clusters() 297, 299
  get_null_illegal_pointers() 299
  get_number() 299
  get_readlock_timeout() 299
  get_transient_segment() 299
  get_writelock_timeout() 299
  is_deleted() 300
  is_empty() 300
  lock_into_cache() 300
  of() 300
  return_memory() 300
  set_access_control() 300
  set_application_info() 301
  set_check_illegal_pointers() 301
  set_comment() 301
  set_fetch_policy() 302
  set_lock_whole_segment() 303
  set_null_illegal_pointers() 304
  set_readlock_timeout() 304
  set_size() 305
  set_writelock_timeout() 305
  size() 305
  unlock_from_cache() 300, 305
  unused_space() 305
~os_segment_access()
  os_segment_access, defined by 310
os_segment_access()
  os_segment_access, defined by 308
os_segment_access, the class 307–310
  get_default() 307

get_primary_group() 308
no_access 308
operator =() 308
~os_segment_access() 310
os_segment_access() 308
read_access 309
read_write_access 309
set_default() 309
set_primary_group() 309
os_segment_access_delete() 520
os_segment_access_get_default() 520
os_segment_access_get_primary_group() 520
os_segment_access_set_default() 520
os_segment_access_set_primary_group() 520
os_segment_allow_external_pointers() 513
os_segment_count_objects() 513
os_segment_create_object_cluster() 513
os_segment_database_of() 513
os_segment_destroy() 513
os_segment_external_pointer_status() 514
os_segment_get_access_control() 514
os_segment_get_all_object_clusters() 514
os_segment_get_application_info() 514
os_segment_get_check_illegal_pointers() 514
os_segment_get_comment() 514
os_segment_get_database_references() 515
os_segment_get_fetch_policy() 515
os_segment_get_lock_whole_segment() 515
os_segment_get_n_object_clusters() 515
os_segment_get_null_illegal_pointers() 515
os_segment_get_number() 515
os_segment_get_readlock_timeout() 516
os_segment_get_size() 516
os_segment_get_transient_segment() 516
os_segment_get_writelock_timeout() 516
os_segment_is_empty() 516
os_segment_lock_into_cache() 516
os_segment_of() 516
os_segment_return_memory() 517
os_segment_set_access_control() 517
os_segment_set_application_info() 517
os_segment_set_check_illegal_pointers() 517
os_segment_set_comment() 517
os_segment_set_fetch_policy() 518
os_segment_set_lock_whole_segment() 518
os_segment_set_null_illegal_pointers() 518
os_segment_set_readlock_timeout() 518
os_segment_set_size() 518
os_segment_set_writelock_timeout() 518
os_segment_size() 519
os_segment_unlock_from_cache() 519
os_server, the class 311–313
connection_is_broken() 312
disconnect() 312
get_databases() 312
get_host_name() 312
get_n_databases() 313
is_failover() 313
reconnect() 313
os_server_connection_is_broken() 521
os_server_disconnect() 521
os_server_get_databases() 521
os_server_get_host_name() 521
os_server_get_n_databases() 521
os_server_reconnect() 522
os_size_options_allocate 430
os_size_options_delete() 430
os_store(), the global function 374
os_str_conv()
os_str_conv, defined by 316
os_str_conv, the class 314–317
change_mapping() 314
convert() 315
get_converted_size() 315
os_str_conv() 316

os_subscription()
  os_subscription, defined by 318
os_subscription, the class 318–320
  assign() 319
  get_database() 319
  os_subscription() 318
os_subscription_assign_database() 503
os_subscription_assign_object_cluster() 503
os_subscription_assign_reference() 503
os_subscription_assign_segment() 503
os_subscription_free() 503
os_subscription_free_array() 503
os_subscription_get_database() 504
os_subscription_new() 502
os_subscription_new_array() 503
os_subscription_sizeof() 503
os_svr_stat_client_info_delete() 430
os_svr_stat_client_meters_delete() 430
os_svr_stat_client_process_delete() 430
os_svr_stat_client_state_delete() 431
os_svr_stat_delete() 431
os_svr_stat_svr_header_delete() 431
os_svr_stat_svr_meters_delete() 431
os_svr_stat_svr_parameters_delete() 431
os_svr_stat_svr_rusage_delete() 431
os_template, the class 321–322
  Function 321
  get_args() 321
  get_kind() 321
  is_unspecified() 321
  operator const os_type_template&() 322
  operator os_type_template&() 322
  set_args() 322
  Type 322
os_template_actual_arg, the class 323–324
  get_kind() 323
  operator const os_literal_template_actual_arg&() 323
  operator const os_type_template_actual_arg&() 323
  operator os_literal_template_actual_arg&() 323
  operator os_type_template_actual_arg&() 323
os_template_actual_arg_get_kind() 490
os_template_formal_arg, the class 325–326
  get_kind() 325
  get_name() 325
  operator const os_template_type_formal&() 325
  operator const os_template_value_formal&() 325
  operator os_template_type_formal&() 325
  operator os_template_value_formal&() 326
  set_name() 326
os_template_formal_arg_get_kind() 490
os_template_formal_arg_get_name() 490
os_template_formal_arg_set_name() 490
os_template_formal_create() 491
os_template_get_args() 491
os_template_get_kind() 491
os_template_instantiation, the class 327–328
  create() 327
  get_args() 327
  get_template() 327
  set_args() 327
  set_template() 328
os_template_instantiation_create() 491
os_template_instantiation_set_args() 491
os_template_instantiation_set_template() 491
os_template_is_unspecified() 492
os_template_set_args() 492
os_template_type_formal, the class 329
  create() 329
os_template_value_formal, the class 330
  create() 330
  get_type() 330
  set_type() 330

os_template_value_formal_create() 492
os_template_value_formal_get_type() 492
os_template_value_formal_set_type() 492
os_tmplt_cast_to_os_type_tmplt() 492
os_transaction, the class 331–338
  abort() 331, 380
  abort_only 331, 379
  abort_top_level() 332, 380
  begin() 332
  checkpoint() 334
  checkpoint_in_progress() 335
  commit() 335
  exception_status 335
  get_current() 335
  get_max_retries() 336
  get_name() 336
  get_parent() 336
  get_scope() 336
  get_type() 336
  is_aborted() 336
  is_committed() 336
  max_retries 380
  read_only 338, 379
  set_max_retries() 338
  set_name() 338
  top_level() 338
  update 338, 379
os_transaction_abort() 523
os_transaction_abort_top_level() 523
os_transaction_begin() 523
os_transaction_begin_named() 523
os_transaction_commit() 523
os_transaction_get_abort_handle() 523
os_transaction_get_current() 524
os_transaction_get_exception_status() 524
os_transaction_get_max_retries() 524
os_transaction_get_name() 524
os_transaction_get_parent() 524
os_transaction_get_type() 524
os_transaction_hook, the class 339–340
  after_begin 339
  after_commit 339
  before_abort 339
  before_commit 339
  before_retry 339
  register_hook() 339
os_transaction_is_aborted() 524
os_transaction_is_committed() 525
os_transaction_set_max_retries() 525
os_transaction_set_name() 525
os_transaction_top_level() 525
os_transaction-hook, the class
  deregister_hook() 340
os_transformer_binding()
  os_transformer_binding, defined by 341
os_transformer_binding, the class 341
  os_transformer_binding() 341
os_transformer_binding_create() 551
os_transformer_binding_destroy() 551
os_type, the class 342–353
  Anonymous_indirect 342
  Array 342
  Char 342
  Class 342
  Double 342
  Enum 342
  Float 343
  Function 343
  get_alignment() 343
  get_enclosing_class() 343
  get_kind() 343
  get_kind_string() 343
  get_size() 344
  get_string() 344
  Instantiated_class 344
  Integer 344
  is_class_type() 344
  is_const() 344
  is_indirect_type() 344
  is_integral_type() 345
  is_pointer_type() 344
  is_real_type() 345

is_unspecified() 345
is_volatile() 345
Long_double 345
Named_indirect 346
operator const os_anonymous_indirect_type&() 346
operator const os_array_type&() 346
operator const os_class_type&() 346
operator const os_enum_type&() 346
operator const os_function_type&() 346
operator const os_instantiated_class_type&() 346
operator const os_integral_type&() 347
operator const os_named_indirect_type&() 347
operator const os_pointer_to_member_type&() 347
operator const os_pointer_type&() 347
operator const os_real_type&() 347
operator const os_reference_type&() 347
operator const os_type_type&() 347
operator const os_void_type&() 348
operator os_anonymous_indirect_type&() 348
operator os_array_type&() 348
operator os_class_type&() 348
operator os_enum_type&() 348
operator os_function_type&() 348
operator os_instantiated_class_type&() 348
operator os_integral_type&() 348
operator os_named_indirect_type&() 349
operator os_pointer_to_member_type&() 349
operator os_pointer_type&() 349
operator os_real_type&() 349
operator os_reference_type&() 349
operator os_type_type&() 349
operator os_void_type&() 349
Pointer 350
Pointer_to_member 350
Reference 350
set_alignment() 350
set_size() 350
Signed_char 350
Signed_long 350
Signed_short 350
strip_indirect_types() 350
Type 352
type_at() 352
type_containing() 352
Undefined 352
Unsigned_char 352
Unsigned_integer 353
Unsigned_long 353
Unsigned_short 353
Void 353
os_type_cast_to_os_anonymous_indirect_type() 493
os_type_cast_to_os_array_type() 493
os_type_cast_to_os_class_type() 493
os_type_cast_to_os_enum_type() 493
os_type_cast_to_os_function_type() 493
os_type_cast_to_os_instantiated_class_type() 494
os_type_cast_to_os_integral_type() 494
os_type_cast_to_os_named_indirect_type() 494
os_type_cast_to_os_pointer_to_member_type() 494
os_type_cast_to_os_pointer_type() 494
os_type_cast_to_os_real_type() 494
os_type_cast_to_os_reference_type() 495
os_type_cast_to_os_type_type() 495
os_type_cast_to_os_void_type() 495
os_type_get_alignment() 495
os_type_get_enclosing_class() 495
os_type_get_kind() 495
os_type_get_kind_string() 496
os_type_get_size() 496
os_type_get_string() 496
os_type_is_const() 496

os_type_is_integral_type() 496
os_type_is_real_type() 496
os_type_is_unspecified() 497
os_type_is_volatile() 497
os_type_set_alignment() 497
os_type_set_size() 497
os_type_strip_indirect_types() 497
os_type_template, the class 354
  create() 354
  get_type() 354
  set_type() 354
os_type_template_actual_arg() 497
os_type_template_actual_arg, the class 355
  create() 355
  get_type() 355
  set_type() 355
os_type_template_actual_arg_get_type() 497
os_type_template_actual_arg_set_type() 498
os_type_template_create() 498
os_type_template_get_type() 498
os_type_template_set_type() 498
os_type_tmplt_cast_to_os_tmplt() 498
os_type_type, the class 356
os_type_type_at() 498
os_type_type_cast_to_os_type() 499
os_type_type_containing() 499
os_type_type_create() 499
os_typed_pointer_void, the class 357
  get_type() 357
  operator void*() 357
os_typespec()
  os_typespec, defined by 362
os_typespec, the class 358–362
  get_char() 358
  get_double() 358
  get_float() 358
  get_int() 359
  get_long() 359
  get_long_double() 359
  get_name() 359
  get_pointer() 359
  get_short() 360
  get_signed_char() 360
  get_signed_int() 360
  get_signed_long() 360
  get_signed_short() 361
  get_unsigned_char() 361
  get_unsigned_int() 361
  get_unsigned_long() 361
  get_unsigned_short() 361
  operator ==() 362
  os_typespec() 362
os_typespec_equal() 526
os_typespec_get_char() 526
os_typespec_get_double() 526
os_typespec_get_float() 526
os_typespec_get_int() 526
os_typespec_get_long() 526
os_typespec_get_long_double() 526
os_typespec_get_name() 527
os_typespec_get_pointer() 527
os_typespec_get_short() 527
os_typespec_get_signed_char() 527
os_typespec_get_signed_int() 527
os_typespec_get_signed_long() 527
os_typespec_get_signed_short() 527
os_typespec_get_unsigned_char() 527
os_typespec_get_unsigned_int() 527
os_typespec_get_unsigned_long() 528
os_typespec_get_unsigned_short() 528
os_typespec_name_is() 528
os_unsigned_int32 7
os_verifydb_options_allocate 432
os_verifydb_options_delete() 432
os_void_type, the class 363
  create() 363
os_void_type_cast_to_os_type() 499
os_void_type_create() 499
~os_with_mapped()
  os_with_mapped, defined by 364

os_with_mapped()
  os_with_mapped, defined by 364
os_with_mapped, the class 364
  ~os_with_mapped() 364
  create() 364
osprmgc()
  os_dbutil, defined by 117
ossize()
  os_dbutil, defined by 113
OSU_DATABASE
  os_rawfs_entry, defined by 201
OSU_DIRECTORY
  os_rawfs_entry, defined by 201
OSU_LINK
  os_rawfs_entry, defined by 201
osverifydb()
  os_dbutil, defined by 114
osverifydb_one_segment()
  osdbutil, defined by 116

P

parameter
  referent type 203, 215, 224, 237, 247, 257
permissions 307
Persistent
  os_member_variable, defined by 167
persistent **delete** 366
persistent **new** 367
persistent storage
  reclaiming using the **delete** operator 366
persistent unions 390
persistently allocated objects
  and clustering 367
  creating 367
Pointer
  os_type, defined by 350
Pointer_to_member
  os_type, defined by 350
pointer_to_void_os_fetch() 441
pointer_to_void_os_store() 442
pointers
  from transient to persistent memory 367
  to memory in another database 367
  validity 367
Private
  os_base_class, defined by 55
  os_member, defined by 157
propagate_log()
  objectstore, defined by 30
Protected
  os_base_class, defined by 55
  os_member, defined by 157
Public
  os_base_class, defined by 55
  os_member, defined by 158

Q

queue_status()
  os_notification, defined by 180

R

random access 95, 302
read_access
  os_segment_access, defined by 309
read_only
  os_transaction, defined by 338, 379
read_write_access
  os_segment_access, defined by 309
read-only transactions 379
receive()
  os_notification, defined by 181
reclaiming storage
  using the **delete** operator 366
reclassification functions 275
reconnect()
  os_server, defined by 313
reconnecting to Servers 311
Reference
  os_type, defined by 350
referent type 203, 215, 224, 237, 247, 257

register_()
  os_DLL_finder, defined by 128
register_hook()
  os_transaction_hook, defined by 339
Regular
  os_member_variable, defined by 167
rehost_all_links()
  os_dbutil, defined by 117
rehost_link()
  os_dbutil, defined by 117
Relationship
  os_member, defined by 158
release()
  os_address_space_marker, defined by 47
  os_retain_address, defined by 269
release_maintenance()
  objectstore, defined by 30
release_major()
  objectstore, defined by 30
release_minor()
  objectstore, defined by 30
release_name()
  objectstore, defined by 31
release_persistent_addresses()
  objectstore, defined by 31
remote schemas 75
remove()
  os_dbutil, defined by 118
  os_dynamic_extent, defined by 133
remove_required_DLL_identifier()
  os_database, defined by 91
rename()
  os_dbutil, defined by 118
reset()
  os_mop, defined by 173
resolve()
  os_Reference, defined by 208
  os_reference, defined by 214
  os_Reference_local, defined by 218
  os_reference_local, defined by 223
  os_Reference_protected, defined by 230
  os_reference_protected, defined by 236
  os_Reference_protected_local, defined by 241
  os_reference_protected_local, defined by 246
  os_Reference_this_DB, defined by 251
  os_reference_this_DB, defined by 256
  os_Reference_transient, defined by 261
  os_reference_transient, defined by 265
retain()
  os_address_space_marker, defined by 47
retain_persistent_addresses()
  objectstore, defined by 31, 203, 209, 215, 220, 224, 231, 237, 242
retaining()
  os_retain_address, defined by 269
retries, transaction 380
retrieving data member address using **os_fetch_address** 373
retrieving data member value using **os_fetch()** 370
return_all_pages()
  objectstore, defined by 31
return_memory()
  objectstore, defined by 31
  os_segment, defined by 300
rmdir()
  os_dbutil, defined by 118
RPC exceptions 581

S

schema evolution
  exceptions 579
  reclassification function 275
  transformer functions 274
schema evolution exceptions 579
schema keys 36
schemas
  remote 75

segment_number
  os_pathname_and_segment_number, defined by 191
segment_of()
  os_object_cluster, defined by 187
segment-level access control 307
segments
  deleting 297
  initial 82, 94
  transient 300
sequential access 96, 302
Servers
  reconnecting to 311
set()
  os_object_cursor, defined by 190
set_access()
  os_base_class, defined by 55
  os_member, defined by 158
set_access_control()
  os_segment, defined by 300
set_access_hooks()
  os_database, defined by 91
set_access_of_get_os_typespec_function()
  os_class_type, defined by 66
set_alignment()
  os_type, defined by 350
set_always_ignore_illegal_pointers()
  objectstore, defined by 32
set_always_null_illegal_pointers()
  objectstore, defined by 32
set_application_info()
  os_database, defined by 93
  os_segment, defined by 301
set_application_schema_path()
  os_dbutil, defined by 119
set_application_schema_pathname()
  objectstore, defined by 32
set_arg_list()
  os_function_type, defined by 143
set_arg_list_kind()
  os_function_type, defined by 143
set_args()
  os_template, defined by 322
  os_template_instantiation, defined by 327
set_auto_open_mode()
  objectstore, defined by 33
set_autoload_DLLs_function()
  objectstore, defined by 32
set_base_classes()
  os_class_type, defined by 66
set_base_member()
  os_access_modifier, defined by 44
set_cache_file()
  objectstore, defined by 34
set_cache_size()
  objectstore, defined by 35
set_call_linkage()
  os_member_function, defined by 161
set_check_illegal_pointers()
  objectstore, defined by 35
  os_database, defined by 93
  os_segment, defined by 301
set_class()
  os_base_class, defined by 56
set_class_kind()
  os_class_type, defined by 66
set_client_name()
  objectstore, defined by 35
  os_dbutil, defined by 119
set_comment()
  os_segment, defined by 301
set_commseg_size()
  objectstore, defined by 35
set_current_schema_key()
  objectstore, defined by 36
set_declares_get_os_typespec_function()
  os_class_type, defined by 67
set_default()
  os_segment_access, defined by 309
set_default_check_illegal_pointers()
  os_database, defined by 94

set_default_lock_whole_segment()
os_database, defined by 94
set_default_null_illegal_pointers()
os_database, defined by 94
set_default_segment()
os_database, defined by 94
set_default_segment_size()
os_database, defined by 95
set_defines_virtual_functions()
os_class_type, defined by 67
set_dispatch_table_pointer_offset()
os_class_type, defined by 67
set_element_type()
os_array_type, defined by 52
set_enclosing_namespace()
os_namespace, defined by 175
set_enumerators()
os_enum_type, defined by 136
set_eviction_batch_size()
objectstore, defined by 36
set_eviction_pool_size()
objectstore, defined by 37
set_evolved_schema_db_name()
os_schema_evolution, defined by 284
set_fetch_policy()
os_database, defined by 95
os_segment, defined by 302
set_handle_transient_faults()
objectstore, defined by 37
set_has_constructor()
os_class_type, defined by 67
set_has_destructor()
os_class_type, defined by 67
set_ignore_illegal_pointers()
os_schema_evolution, defined by 284
set_illegal_pointer_handler()
os_schema_evolution, defined by 284
set_incremental_schema_installation()
objectstore, defined by 37
os_database, defined by 96
set_indirect_virtual_base_classes()
os_class_type, defined by 67
set_instantiation()
os_instantiated_class_type, defined by 145
set_introduces_virtual_functions()
os_class_type, defined by 67
set_is_abstract()
os_class_type, defined by 67
set_is_const()
os_anonymous_indirect_type, defined by 49
os_member_function, defined by 161
set_is_forward_definition()
os_class_type, defined by 68
set_is_inline()
os_member_function, defined by 162
set_is_overloaded()
os_member_function, defined by 162
set_is_persistent()
os_class_type, defined by 68
set_is_pure_virtual()
os_member_function, defined by 162
set_is_static()
os_member_function, defined by 162
set_is_virtual()
os_member_function, defined by 162
set_is_volatile()
os_anonymous_indirect_type, defined by 49
os_member_function, defined by 162
set_literal()
os_literal_template_actual_arg, defined by 151
set_local_references_are_db_relative()
os_schema_evolution, defined by 286
set_locator_file()
objectstore, defined by 37
set_lock_whole_segment()
os_database, defined by 97
os_segment, defined by 303

set_log_file()
  objectstore, defined by 38
set_mapped_communications()
  objectstore, defined by 38
set_max_retries()
  os_transaction, defined by 338
set_members()
  os_class_type, defined by 68
  os_namespace, defined by 175
set_name()
  os_class_type, defined by 68
  os_enum_type, defined by 136
  os_enumerator_literal, defined by 137
  os_member_function, defined by 162
  os_member_variable, defined by 168
  os_named_indirect_type, defined by 174
  os_namespace, defined by 175
  os_pointer_literal, defined by 192
  os_template_formal_arg, defined by 326
  os_transaction, defined by 338
set_namespace()
  os_member_namespace, defined by 164
set_new_id()
  os_database, defined by 97
set_null_illegal_pointers()
  objectstore, defined by 38
  os_database, defined by 97
  os_segment, defined by 304
set_number_of_elements()
  os_array_type, defined by 53
set_obsolete_index_handler()
  os_schema_evolution, defined by 286
set_obsolete_query_handler()
  os_schema_evolution, defined by 286
set_offset()
  os_base_class, defined by 56
  os_member_variable, defined by 169
set_opt_cache_lock_mode()
  objectstore, defined by 39
  os_database, defined by 98
set_pragmas()
  os_class_type, defined by 68
  os_enum_type, defined by 136
set_primary_group()
  os_segment_access, defined by 309
set_queue_size()
  os_notification, defined by 182
set_readlock_timeout()
  objectstore, defined by 39
  os_database, defined by 98
  os_segment, defined by 304
set_reconnect_timeout_and_interval()
  os_failover_server, defined by 140
set_relative_directory()
  os_database, defined by 98
set_reserve_as_mode()
  objectstore, defined by 39
set_retain()
  os_retain_address, defined by 269
set_return_type()
  os_function_type, defined by 143
set_schema_database()
  os_database, defined by 99
set_schema_database_pathname()
  os_schema_handle, defined by 291
  os_schema_info, defined by 293
set_simple_auth_ui()
  objectstore, defined by 39
set_size()
  os_field_member_variable, defined by 141
  os_segment, defined by 305
  os_type, defined by 350
set_source_position()
  os_class_type, defined by 68
  os_enum_type, defined by 136
  os_member_function, defined by 162
  os_member_variable, defined by 169
  os_named_indirect_type, defined by 174
set_storage_class()
  os_member_variable, defined by 169

set_target_class()
os_pointer_to_member_type, defined by 193
set_target_type()
os_indirect_type, defined by 144
os_pointer_type, defined by 194
set_task_list_file_name()
os_schema_evolution, defined by 287
set_template()
os_template_instantiation, defined by 328
set_thread_locking()
objectstore, defined by 40
set_transaction_priority()
objectstore, defined by 41
set_transient_delete_function()
objectstore, defined by 41
set_type()
os_member_function, defined by 163
os_member_type, defined by 165
os_member_variable, defined by 169
os_pointer_literal, defined by 192
os_template_value_formal, defined by 330
os_type_template, defined by 354
os_type_template_actual_arg, defined by 355
set_unhandled_exception_hook()
tix_exception, defined by 561
set_value()
os_database_root, defined by 104
os_enumerator_literal, defined by 137
set_virtual_base_class_no_pointer()
os_base_class, defined by 56
set_virtual_base_class_pointer_offset()
os_base_class, defined by 56
set_virtuals_redefined()
os_base_class, defined by 56
set_writelock_timeout()
objectstore, defined by 42
os_database, defined by 99
os_segment, defined by 305
short_os_fetch() 442
short_os_store() 442
shutdown()
objectstore, defined by 42
signal()
objectstore_exception, defined by 559
Signed_char
os_type, defined by 350
signed_char_os_fetch() 442
signed_char_os_store() 442
Signed_long
os_type, defined by 350
Signed_short
os_type, defined by 350
size()
os_database, defined by 99
os_segment, defined by 305
size_in_sectors()
os_database, defined by 100
stat()
os_dbutil, defined by 119
Static
os_member_variable, defined by 167
strip_indirect_types()
os_type, defined by 350
Struct
os_class_type, defined by 68
subscribe()
os_notification, defined by 182
svr_checkpoint()
os_dbutil, defined by 119
svr_client_kill()
os_dbutil, defined by 120
svr_ping()
os_dbutil, defined by 120
svr_shutdown()
os_dbutil, defined by 121
svr_stat()
os_dbutil, defined by 121

T

task_list()
  os_schema_evolution, defined by 287
time_created()
  os_database, defined by 100
tix_context, the class 563
tix_exception, the class
  set_unhandled_exception_hook() 561
tix_exception_message() 552
tix_exception_parent() 552
tix_exception_signal() 552
tix_exception_signal_val() 553
tix_exception_vsignal() 553
tix_handler, the class
  get_current() 561
  get_exception() 561
  get_report() 561
  get_report0() 562
  get_value() 560, 562
tix_handler_get_current() 553
tix_handler_get_exception() 553
tix_handler_get_report() 553
tix_handler_get_value() 553
top_level()
  os_transaction, defined by 338
transaction macros
  begin and end 379
transactions
  abort-only 379
  aborts 380
  and deadlock 380
  locking 379
  nested
    and locking 379
  read-only 379
  retry 380
  system aborts 380
  update 379
  user aborts 380
transfer mode 95, 302
transformer functions 274
transient segments 300
transiently allocated objects
  and clustering 367
  creating 367
two-phase commit 381
Type
  os_member, defined by 158
  os_template, defined by 322
  os_type, defined by 352
type_at()
  os_type, defined by 352
type_containing()
  os_type, defined by 352

U

Undefined
  os_type, defined by 352
Union
  os_class_type, defined by 69
unions, persistent 390
Unknown
  os_function_type, defined by 143
unload_DLL()
  objectstore, defined by 43
unlock_from_cache()
  os_segment, defined by 300, 305
unregister()
  os_DLL_finder, defined by 128
Unsigned_char
  os_type, defined by 352
unsigned_char_os_fetch() 443
unsigned_char_os_store() 443
unsigned_int_os_fetch() 443
unsigned_int_os_store() 443
Unsigned_integer
  os_type, defined by 353
Unsigned_long
  os_type, defined by 353
unsigned_long_os_fetch() 443
unsigned_long_os_store() 443

Unsigned_short
    os_type, defined by 353
unsigned_short_os_fetch() 444
unsigned_short_os_store() 444
unsubscribe()
    os_notification, defined by 183
unused_space()
    os_segment, defined by 305
update
    os_transaction, defined by 338, 379
update transactions 379
utility API 106

V

Variable
    os_function_type, defined by 143
    os_member, defined by 158
virtual_base_class_has_pointer()
    os_base_class, defined by 56
virtuals_redefined()
    os_base_class, defined by 56
Void
    os_type, defined by 353
vsignal()
    objectstore_exception, defined by 560

W

wchar_t_os_fetch() 444
wchar_t_os_store() 444
which_product()
    objectstore, defined by 43