OBJECTSTORE

C++ API User Guide

RELEASE 5.1

March 1998

ObjectStore C++ API User Guide

ObjectStore Release 5.1 for all platforms, March 1998

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

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

Contents

Preface

Purpose

The *ObjectStore C++ API User Guide* describes how to use the basic C++ programming interface to ObjectStore to create database applications, using the fundamental features of ObjectStore. This book supports ObjectStore Release 5.1.

This publication’s companion volume, the *ObjectStore Advanced C++ API User Guide*, provides descriptions of the more complex features in ObjectStore.

Audience

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

Scope

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

How This Book Is Organized

The book begins with a chapter on basic ObjectStore concepts. Each chapter after that covers a broad area of database functionality, including persistence, transactions, notifications, query processing, integrity control, access control, and schema evolution, among others.

In contrast to the *ObjectStore C++ API Reference* and *ObjectStore Collections C++ API Reference* manuals, both of which are organized alphabetically, the two ObjectStore user guides are organized functionally. This manual, the *ObjectStore C++ API User Guide*, describes fundamental functions and macros. The *ObjectStore Advanced C++ API User Guide* contains advanced features; it also organizes the ObjectStore API into groups of related functions and macros.

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

ObjectStore C++ Release 5.1 Documentation

The ObjectStore Release 5.1 documentation is chiefly distributed 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 of 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 Internet Explorer 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.

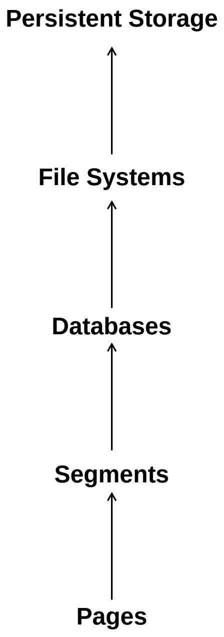
Chapter 1
ObjectStore Concepts

This chapter introduces the basic concepts you need to understand in order to use ObjectStore successfully. The information is organized as conceptual overviews of the following topics:

Persistent Storage

Persistent data is data that survives beyond the lifetime of the process that created it. ObjectStore stores persistent data in stable storage in databases, typically disks. There are two kinds of databases. A *file database* is a regular operating system file. A *rawfs database* resides in an ObjectStore file system managed by an ObjectStore Server. Rawfs databases are discussed further on page 5.

Each database is made up of *segments*, which are variable-sized regions of memory that can be used as the unit of transfer from persistent storage to program, or *transient*, memory. Each segment, in turn, is made up of pages. A specified number of pages can be used instead of segments as the unit of transfer to program memory.



ObjectStore Processes

ObjectStore

ObjectStore applications require two auxiliary processes for application execution — an *ObjectStore Server* and a *Cache Manager*. A Server handles access to ObjectStore databases, including storage and retrieval of persistent data. When ObjectStore is installed, the system administrator typically arranges for each Server to start when its host machine boots. A single application can use several databases, including databases on different file systems, handled by different Servers. Most users never have to worry about starting and stopping Servers.

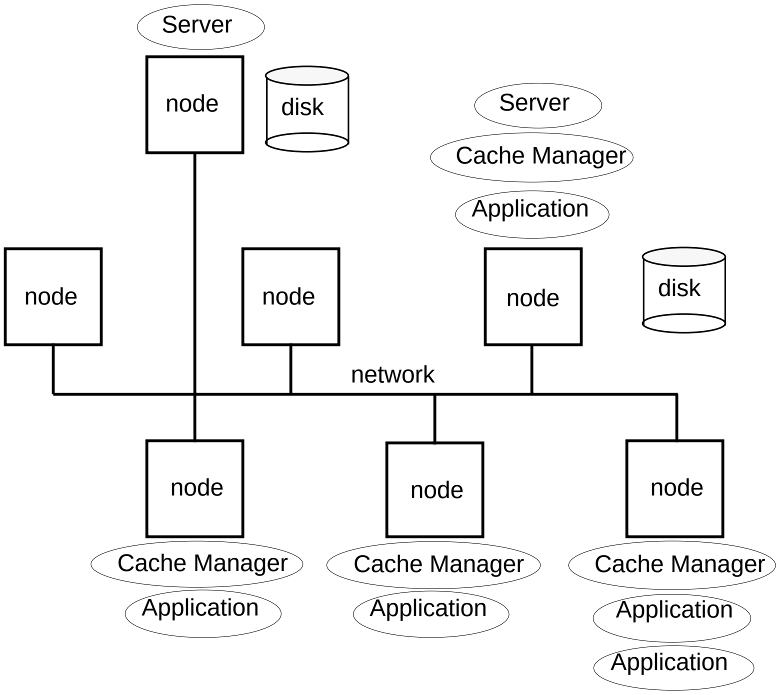
A Cache Manager is started automatically when an ObjectStore application starts. The cache manager is a daemon that runs on the machine running the client application. Its function is to respond to Server requests as a stand-in for the client application, in order to participate in the management of the application's *client cache.* The client cache is the local holding area for data mapped or waiting to be mapped into virtual memory.

If additional ObjectStore applications are started on the same machine, the same Cache Manager functions similarly for these applications as well. Although the same machine can run several ObjectStore applications at once, only a single Cache Manager is ever running on a given machine. As with Servers, most users never have to worry about starting or stopping Cache Managers.

ObjectStore / Single

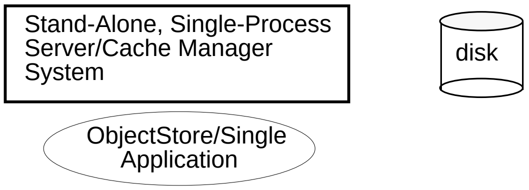
ObjectStore / Single applications have combined the Server and Cache Manager functions into one process. The diagrams that follow point out the distinctions between the ObjectStore and ObjectStore / Single implementations. It is important to remember that although the implementation differs, the functions performed by the Server and Cache Manager are preserved in ObjectStore / Single. For more information, see *ObjectStore Building C++ Interface Applications*, Chapter 6, Working with ObjectStore / Single.

ObjectStore processes



Each site has one or more Servers to handle persistent data. Each node running one or more ObjectStore applications has a Cache Manager to handle application data.

ObjectStore / Single process


Stand-Alone, Single-Process Server/Cache Manager System
disk
ObjectStore/Single Application

The nonnetworked ObjectStore / Single application provides the same functions, but the two processes have been combined into one.

Rawfs Databases

Using ObjectStore, you have the option of storing some or all of your databases in ObjectStore file systems managed by ObjectStore Servers instead of storing databases as regular files managed by the operating system. Each ObjectStore file system, known as a *rawfs*, is either a raw partition or an operating system file. For information on setting up and managing a rawfs, see *ObjectStore Management*, Chapter 1, Overview of Managing ObjectStore, Managing the Rawfs.

Each rawfs provides a separate name space and directory hierarchy. Rawfs directories form hierarchical structures just as operating system directories do. But rawfs directory hierarchies are independent of the operating system directory hierarchies.

Each ObjectStore Server can manage a hierarchy of rawfs directories, and maintains permission modes, creation dates, owners, and groups for each entry. There can be several independent rawfs directory hierarchies at a given site, each managed by a different Server, and the same application can use databases in different hierarchies.

ObjectStore / Single

Currently, ObjectStore / Single supports only file databases, not rawfs databases.

ObjectStore Memory Mapping Architecture

With ObjectStore, data is transferred between database memory and program memory completely automatically in a manner transparent to the user. ObjectStore detects any reference in a running program to persistent data, and automatically transfers the page containing the referenced data (possibly together with adjacent pages) across the network to the application’s cache. Then the page containing the referenced data is mapped into virtual memory.

Sometimes the referenced data is already in the client cache (because data in the same pages was already used, and the required page was not swapped out of the cache), and all that is required is the virtual memory mapping. Sometimes the data is already mapped into virtual memory (because data on the same page was already used in the current transaction) and then nothing additional is required to access it. Once data has been mapped into virtual memory, access to it is as fast as access to regular, transient data.

The paragraphs that follow provide a summary of how the transfer of data between persistent and transient memory is handled.

ObjectStore achieves the combination of transparency and efficiency with a unique memory mapping architecture. All data is stored in an ObjectStore database in its native C++ format. All pointers in a database take the form of regular virtual memory pointers. The value of a pointer in a given segment is the segment’s *pseudoaddress* for the object the pointer refers to.

Pseudoaddresses

A pseudoaddress is the identifier a segment uses for an object pointed to by that segment. Two pointers (pseudoaddresses) in the same segment have the same value if and only if they refer to the same object. But two pointers (pseudoaddresses) in different segments might have the same value and yet refer to different objects. And two pointers (pseudoaddresses) in different segments might have different values and yet refer to the same object. This is what makes *relocation* necessary.

Relocation is the process of changing the pointers on a page of data as it is mapped into the client cache or unmapped from the client cache. Here is how relocation works.

Persistent relocation map (PRM)

Each segment has an associated persistent table, the *persistent relocation map* (PRM), that allows determination of an object's on-disk location (that is, database name, segment number, and offset), given its pseudoaddress. The PRM does not actually have an entry for each different pseudoaddress. Instead, each entry covers a range of pseudoaddresses.

When a page of data is mapped into the cache, all pseudoaddresses on the page must be converted into virtual addresses within the process's persistent address space region. Within a transaction, a single unified mapping (across all segments) that establishes the translation between database location (again, database name, segment number, and offset) to virtual address is built up. Also, for each segment, a transient mapping that establishes the direct bidirectional translation between pseudoaddresses and virtual addresses is built up. The manner in which these mappings are built depends on whether the segment is using

- Immediate address-space assignment
- Deferred address-space assignment

With immediate assignment, a segment's whole PRM is incorporated into the unified mapping, and the whole pseudoaddress-to-virtual-address mapping for that segment is built prior to the first time any page in that segment is mapped into the cache.

With deferred assignment, the unified mapping and the pseudoaddress-to-virtual-address mapping are augmented from the segment's PRM as necessary to translate each pseudoaddress encountered during the relocation. In both cases, each pointer (pseudoaddress) on the page being relocated is changed to the virtual address determined by its on-disk location and the transient mappings described above. Once inbound relocation is complete, the page is mapped into virtual memory at the location assigned it by the unified mapping.

Outbound relocation

When a modified page is written to the database, *outbound relocation* is performed on it, which causes its pointers to be changed back to their original pseudoaddresses. Any new pointers to on-disk locations not yet assigned pseudoaddresses cause the page's PRM to be augmented with new entries.

Transient relocation map (TRM)

In some cases, ObjectStore skips outbound relocation, because it knows that the page's pointers and the corresponding pseudoaddresses are the same (this is determined by ObjectStore during inbound relocation). In such a case, the PRM is augmented with entries to accommodate all on-disk locations currently in the in-use *transient relocation map* (TRM).

Advantages of this architecture

Two of the major advantages of this architecture can be described as follows:

- Persistence is specified on a per-instance basis, independent of type. The same type can have both persistent and nonpersistent instances, and the same function can operate on both persistent and nonpersistent data. Moreover, instances of any built-in C++ type (such as **int**) can be designated as persistent. This means that your existing routines, developed for use with transient data, can also be used in ObjectStore applications.
- Pointers are processed at memory speeds. Once an object has been transferred and mapped into virtual memory, all pointers to it are regular virtual memory pointers, and are processed at regular hardware speeds, with none of the overhead associated with *soft* pointer schemes, and no continual checking for database references.

Memory Mapping and Schema Information

For ObjectStore to realize the advantages inherent in this memory mapping architecture, it needs to store *schema information* in each database — that is, it needs to store information in each database about the classes of objects stored there, and the layout of instances of these classes. This allows ObjectStore to identify the locations of pointer fields in each newly retrieved segment (so it can perform relocation).

ObjectStore stores schema information as C++ objects. Classes themselves are not run-time objects in C++ (they cannot, for example, be values of variables or other expressions). So ObjectStore must generate representations of classes in order to manage database memory.

These representations are generated, before link time, for each application that might store information in a database. So, at run time, when the application stores an object in a database, a representation of the object's class is ready to be added to the database's schema along with the object itself; or, if instances of that class are already in the database, the application's class representation is checked against that of the database to make sure they agree.

An application's schema information (generated by the ObjectStore *schema generator*) is stored in two places: a source file and an ObjectStore database. Because you must use the schema generator when building an ObjectStore application, you are making use of ObjectStore's database management capabilities, which are described in *ObjectStore Management*.

Generating Schemas for ObjectStore Applications

Building an ObjectStore application has a step not associated with regular, nondatabase C++ applications: the generation of schema information. This process is performed by the ObjectStore schema generator, and produces both an ObjectStore database, known as the *application schema database*, and an object file, the *application schema source file*.

Input to Schema Generation

The input to schema generation consists of *schema source files*, possibly together with *library schemas*. Schema source files are files you provide. In them, you list those classes whose instances are created and stored in persistent memory by the application, or whose instances serve as *entry points* into persistent memory (see Database Entry Points and Data Retrieval on page 47). The schema source file that you provide should include additional information if you use either ObjectStore *dictionaries* (see Dictionaries on page 139), or *query functions* (see Chapter 5, Queries and Indexes, in the *ObjectStore Advanced C++ API User Guide*).

By performing a *transitive closure operation*, the schema generator determines all those classes reachable by navigation from the classes in the schema source file, and adds information about them to the application schema database.

Library schemas are ObjectStore databases that contain schema information for libraries that store or retrieve persistent data. ObjectStore provides library schemas for its libraries. You can also use the schema generator to generate library schemas for other libraries.

Schema Generator Output

The *application schema source file* is an output file that is created by the ObjectStore schema generator, and must be compiled and linked with your application. This file records the location of the application schema database and the names of the application's virtual function dispatch tables. It also contains discriminant functions for anonymous unions.

For complete instructions on building ObjectStore applications and libraries, see *ObjectStore Building C++ Interface Applications*. In particular, there is a good summary of tasks in Chapter 1, Overview of Building an Application. The process of schema generation itself is discussed in detail in Chapter 3, Generating Schemas, of that publication.

Programming Interface

The ObjectStore C++ interface is designed for the development of C++ and C applications that require database services. Although some of your interaction with ObjectStore takes place from the shell (you issue commands, for example, to create rawfs directories or generate schemas), most of it takes place from within programs.

With ObjectStore and ObjectStore / Single, you can use a variety of C++ compilers, together with the C++ library interface. This is a library of classes whose member functions, data members, and enumerators provide access to database functionality. Also included are global functions, such as an overloading of **operator new()** that allows dynamic allocation of persistent memory for any type of object.

The class templates feature of C++ is included in the ANSI Standard for C++. If you are using a compiler that supports templates, you can use the parameterized versions of some classes in the class library. Using parameterized classes enhances the type safety of your applications.

Chapter 2 Persistence

With ObjectStore, storing persistent data is a lot like storing transient data with plain C++ or C — you allocate memory and assign a value to that memory. The only difference is that if you want to store persistent data, you allocate *persistent memory*. What sort of memory you allocate (persistent or transient) is independent of the type of value you want to store there, so any type of value can be stored in either persistent or transient memory.

The information is organized as follows:

Basic Behaviors

The paragraphs that follow introduce key characteristics and provide references to more detailed discussions about each.

Persistent new() and delete()

You can allocate and initialize persistent memory by using an overloaded C++ **new** operator, supplied by the ObjectStore API. There is also a version of the C++ **delete** operator that you can use to delete persistent objects and free persistent memory. Creation and deletion of persistent objects with **new** and **delete** are described in Persistent new and delete on page 33.

Once you have allocated persistent memory, you can use pointers to this memory in the same way you use any pointers to virtual memory. Pointers to persistent memory, in fact, always take the form of virtual memory pointers. See ObjectStore Memory Mapping Architecture on page 6 for an understanding of how memory mapping works.

Prerequisites to Persistent Access

Before you access persistent memory, you must set the stage by performing a few other operations:

- A database must be *created* or *opened*.
- A *transaction* must be started.
- A *database root* must be retrieved or created.

All these operations are described briefly in this chapter. For more detailed information on transactions, however, see Chapter 3, Transactions, on page 69.

Databases

When you create a persistent object, you create it in a particular database. You specify the database as an argument to persistent **new**. Before any of this, however, you must *create* a database. In subsequent processes, you must *open* the database each time you read from or write to it. These topics are discussed in the following sections:

- Creating Databases on page 20

- Opening Databases with os_database::open() on page 25
- Persistent new and delete on page 33

Clustering

In addition to specifying a new object’s database, you can specify the *segment* (object cluster) within that database in which you want the object stored. By clustering together objects that are expected to be used together by applications, you can improve application performance. Effective clustering reduces both the number of disk and network transfers the applications require and it increases concurrency among applications. Clustering is one of the most important ways of optimizing database performance. Segments and clustering are described in the following sections:

- Basic Clustering on page 45
- Creating Segments on page 60

Additional information on segments and clustering can be found in Chapter 1, Advanced Persistence, of the *ObjectStore Advanced C++ API User Guide*.

Transactions

A program must, before it accesses persistent data, start a *transaction*. While the transaction is in progress, the program’s actions can include reads and writes to persistent objects. The program can then either *commit* or *abort* the transaction at any time.

When a transaction is *committed*, changes made to persistent data during the transaction are made permanent in the database and visible to other processes. These changes are made permanent and visible only if and when the transaction commits. Changes to persistent data are undone or *rolled back* if the transaction in which they were made is *aborted*.

So transactions do two things:

- They mark off code sections whose effects can be undone.
- They mark off functional program areas that are isolated from the changes performed by other processes. From the point of view of other processes, these functional sections execute either

all at once or not at all. That is, other processes do not see the intermediate results.

This latter aspect of transactions is important in preventing concurrency anomalies that can arise from the sharing of persistent data. The former aspect is important in preventing data corruption due to system or network failure.

Transactions are discussed in further detail in Chapter 3, Transactions, on page 69. Additional information can be found in Chapter 2, Advanced Transactions, of the *ObjectStore Advanced C++ API User Guide*.

Roots and Entry-Point Objects

A database root provides a way to give an object a persistent name, allowing the object to serve as an initial *entry point* into persistent memory. When an object has a persistent name, any process can look it up by that name to retrieve it. Once you have retrieved one object, you can retrieve any object related to it by using navigation (that is, following data member pointers), or by a *query* (a query selects all elements of a given collection that satisfy a specified condition — see Chapter 5, Queries and Indexes, in the *ObjectStore Advanced C++ API User Guide*).

Each database typically has a relatively small number of entry point objects (that is, named objects), each of which allows access to a large network or collection of related objects. See

- Establishing Entry Points on page 48
- Retrieving Entry Points on page 50

Requirements for ObjectStore Applications

For a program to use any ObjectStore features, it must include the file **ostore/ostore.hh** and link with the ObjectStore library. For information about these requirements, see ObjectStore Header Files in Chapter 2 of *ObjectStore Building C++ Interface Applications*.

Initializing ObjectStore

Any program using ObjectStore functionality (with certain exceptions, listed below) must first call the static member function **objectstore::initialize()**.

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

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

The following functions are exceptions to this, and *must* be called before **objectstore::initialize()**:

- **objectstore::set_application_schema_pathname()**
- **objectstore::set_cache_size()**
- **objectstore::set_client_name()**

See the entry for the class **objectstore** in the *ObjectStore C++ API Reference* for more information on these functions.

Single-Threaded Applications

If your application does not use multiple threads, you must disable thread locking by issuing the following two calls at the beginning of the transaction:

```
objectstore::set_thread_locking(0) ;
os_collection::set_thread_locking(0) ;
```

For information on thread locking, see Threads and Thread Locking on page 84.

Fault Handler Macros for Multithreaded Applications

On the HP–UX, SGI IRIX, and Digital UNIX platforms, signal handlers are installed strictly on a per-thread basis and are not inherited across **pthread_create** calls. On these platforms, then, you must use the **OS_ESTABLISH_FAULT_HANDLER** and **OS_**

END_FAULT_HANDLER macros at the beginning and end of any thread that performs ObjectStore operations. Note that these macros are only required for threads that use ObjectStore.

See the UNIX-specific information regarding fault handlers in Establishing Fault Handlers in POSIX Thread Environments in Chapter 4, of *ObjectStore Building C++ Interface Applications*.

Creating Databases

You create databases with members of the class **os_database**. This is true for both file databases and ObjectStore rawfs databases.

Creating a Database with os_database::create()

You create a database by calling the static member function **os_ database::create()**. This function also opens a newly created database. You can call this function from either inside or outside a transaction, although, in general, it is best to create databases outside a transaction.

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

Note that, since **os_database::create()** is static, it has no **this** argument.

Database Pathnames

pathname is the name you want the new database to have. It is the only required argument for **os_database::create()**.

When you create file databases, the pathname consists of an operating system pathname. (This book uses UNIX-style database pathnames with slashes (/) as separators, but your operating system might use a different style of pathname.)

```
os_database *db1 = os_database::create( "/ken/parts1" ) ;
```

ObjectStore takes into account local network mount points when interpreting the pathname, so the pathname can refer to a database on a foreign host. Remember that an ObjectStore Server must be running on any host containing an ObjectStore database.

If you want to refer to a file database on a foreign UNIX host for which there is *no* local mount point, you can use a Server host prefix, the name of the foreign host followed by a colon (:), as in **oak:/foo/bar**.

The requisite database pathname syntax differs for rawfs databases. See Rawfs Databases on page 31.

If you supply the pathname of a database that already exists, the exception err_database_exists is signaled at run time (but see Automatic Overwrite, below).

Database Modes

The **mode** argument specifies a *protection mode* (see the **oschmod** utility in *ObjectStore Management*). The mode defaults to **0664**. The mode argument must begin with a zero (**0**) to indicate that it is an octal number. For example:

```
os_database *db1 = os_database::create("/ken/parts1", 0666) ;
```

Automatic Overwrite

It is possible to direct **os_database::create()** to overwrite any existing database with the same name, instead of signaling an exception. You do this by specifying a nonzero value (true) as the value of the optional argument **if_exists_overwrite**.

```
os_database *db1 = os_database::create("/ken/parts1", 0666, 1);
```

This argument defaults to **0** (false). If no database of that name already exists, the third argument has no effect.

Schema Databases

Every ObjectStore database has associated *schema information*; that is, information about the classes of objects the database contains. See Database Schemas on page 55.

When you create a database, if the **schema_database** argument is **0**, schema information is stored in the new database (which you are creating). If **schema_database** is nonzero, the argument is interpreted to be another, extant, database that will be used as the *schema database* for the newly created database. ObjectStore puts all the schema information for the new database in the specified schema 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 is open 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. Consequently, the newly created database cannot accept any updates that require schema information, since no schema information could be written to the schema database.

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.

See also **os_database::get_schema_database()** and **os_ database::set_schema_database()** in the *ObjectStore C++ API Reference*.

Return Value

The static member function **os_database::create()** returns a pointer to an object of type **os_database**. You use this pointer as the **this** argument to other nonstatic member functions of the class **os_database**, such as **os_database::close()** (the function you use to close a database).

The database object is actually *transient*, unlike the database it stands for. That is, it exists only for the duration of the current process. If you copy it into persistent storage, it will be meaningless when retrieved by another process. If you want to record the identity of a database in persistent storage, you should record the database's pathname.

Creating Databases with os_database::open()

Typically, **os_database::create()** is used to create a database, but, under some circumstances, the function **os_database::open()** creates a database. One form of this function requires you to specify the pathname of the database you want to open; you can specify an optional argument to direct the function to create a database if a database with the specified pathname does not exist. See the description of this function in Opening Databases with os_ database::open() on page 25.

Destroying Databases with **os_database::destroy()**

You can destroy a database with the **os_ database::destroy()** function.

```
void destroy() ;
```

This function takes no arguments (other than the **this** argument), and has no return value.

```
os_database* db1;
. . .
db1–>destroy();
```

If the database is open at the time of the call, **destroy()** closes the database before deleting it. Note that to help ensure program portability, you should call the **destroy** function from outside any transaction.

Warning: potential effects on other processes

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

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 still be in the process’s local cache.

If you call this function to destroy a database from within a transaction, be aware 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 destroyed (deleted from the Server on which it resides) at the end of the transaction; in some cases, however, it might be destroyed 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.

Opening Databases with **os_database::open()**

Before you can read or write data, you must open the database in which the data resides. A database is automatically opened when you create it with **os_database::create()**, as described in Creating Databases on page 20.

Opening a Database with os_database::open()

If you want to open a previously created database, you use the function **os_database::open()**.

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

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

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

The first two overloadings return a pointer to the opened database (see Return Value on page 22).

Database Pathnames

If there is no database with the specified pathname, an err_database_not_found exception is signaled. For information about pathnames for file databases, see Database Pathnames on page 20.

Rawfs databases use a somewhat different pathname syntax; see Rawfs Databases on page 31.

Opening Read-Only

The **os_database::open()** function has an optional argument, an **os_boolean** (**int** or **long**, whichever is 32 bits on your platform) that specifies whether the database is to be opened for read only. A nonzero integer indicates read only, and **0** indicates that write access is allowed. The default is

```
os_database *db1 = os_database::open("/ken/parts1", 1) ;
```

If you attempt write access to a database that has been opened for read only, an err_opened_read_only exception is signaled.

Automatic Creation

The optional **create_mode** argument to **os_database::open()** can specify the mode of a new database to be created if a database with the specified pathname does not exist. If this argument is not **0** and a database with the specified name does not exist, instead of signaling an exception, a new database is created with the specified mode (see Database Modes on page 21). This argument defaults to **0**, so if it is not supplied (or if **0** is supplied explicitly), ObjectStore signals an err_database_not_found instead of creating a new database.

```
os_database* db1 = os_database::open("/ken/parts1", 0, 0664);
```

Schema Database

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. See Schema Databases on page 21 for information on how this is handled.

Multiversion Concurrency Control (MVCC)

If you want to use *multiversion concurrency control* (MVCC), which allows you to perform nonblocking reads of a database, you can do so with the following members of **os_database**:

```
void open_mvcc();
```

```
static os_database *open_mvcc(const char *pathname);
```

Multiversion Concurrency Control (MVCC) is described in Chapter 2 of the *ObjectStore Advanced C++ API User Guide*.

Closing Databases with **os_database::close()**

You close a database by calling **os_database::close()**.

```
void close() ;
```

This makes the database’s data inaccessible (assuming you have not performed nested opens — see Nested Database Opens on page 28). If the call to **os_database::close()** occurs inside a transaction, the database is not actually closed until the end of the outermost transaction. That is, the data remains accessible until the outermost transaction terminates.

```
os_database* db1;
. . .
OS_BEGIN_TXN(tx1, 0, os_transaction::update)
    . . .
    db1–>close();
    . . .    /* data in db1 remains accessible */
OS_END_TXN(tx1)

/* end of transaction, so db1 data is now inaccessible */
```

Nested Database Opens

What happens if you open the same database twice, without closing it between the two opens? This is not an error. It simply means that two calls to **os_database::close()** will be required to close the database.

This is because, whenever **os_database::open()** is performed on a database, its *open count* is incremented by one. In order to close a database and make it inaccessible, its open count must reach zero. Each call to **os_database::close()** decrements the open count by one. If the call occurs inside a transaction, it does not actually take effect until the end of the outermost transaction. That is, the open count is not decremented until the outermost transaction terminates.

Nested Opens and Read-Only Access

If you open a database, say, for read / write access (changing the open count from 0 to 1), and then open it again for read-only access (increasing the open count to 2), only read access is allowed until the next call to **os_database::close()** (changing the open count back to 1). In general, the last call to **os_database::open()** that incremented the open count to *n* determines the type of access (read / write or read-only) allowed as long as the open count is *n*.

```
#define TRUE 1

main(){
    os_database* db1;
    . . .
    db1–>open(); /* open count is 1, read/write access */
    . . .
    my_func(db1);
    . . .
    db1–>close(); /* open count is 0, data inaccessible */
}

void my_func (os_database *the_db) {
    the_db–>open(TRUE); /* open count is 2, read-only access */
    . . .
    the_db–>close(); /* open count is 1, read/write access */
}
```

Determining Database Open Status

You can check the status of a database — whether and how it is opened — with members of the class **os_database**.

```
os_boolean is_open() const;

os_boolean is_open_read_only() const;

os_boolean is_open_mvcc() const;
```

These functions return a nonzero integer for true and **0** for false. So, for example, the loop shown below repeatedly calls **close()** on the database **db1** until its open count is zero:

```
os_database* db1;
. . .
while (db1–>is_open())
    db1–>close();
```

Warning against infinite loop

Do not include a loop like this within a transaction. This is because the calls to **os_database::close()** do not take effect until the transaction ends, as noted earlier, so **is_open()** will always return true and your program will contain an infinite loop.

Finding a Specified Database

You can find a database by its pathname using the static member function **os_database::lookup()**, if you simply want to retrieve a pointer to a database without opening it.

static os_database *lookup(const char *pathname, create_mode) ;

If the named database exists, a pointer to the database is returned, but the database remains unopened. If no such database exists and the **create_mode** argument is either **0** or absent, an err_ database_not_found exception is signaled. If **create_mode** is nonzero, however, ObjectStore attempts to create the database, just as **os_database::open()** does (see Opening Databases with os_ database::open() on page 25).

os_database::lookup() is described in the *ObjectStore C++ API Reference*.

Pathname Lookup

You can find the pathname of a database, given a pointer to it, by using the function **os_database::get_pathname()**.

char *get_pathname() const ;

This function returns a **char***, the pathname of the database pointed to by the **this** argument. The **char*** points to an array allocated on the heap by this function, so it is your responsibility to deallocate the array when it is no longer needed.

Rawfs Databases

In addition to file databases, which are regular operating system files contained in operating system directory hierarchies, ObjectStore supports rawfs databases, which are contained in ObjectStore directory hierarchies and managed by ObjectStore Servers.

Rawfs Host Prefix

Pathname arguments in the database functions described in preceding sections (**os_database::create()**, **open()**, and **lookup()**) can designate rawfs databases by including a rawfs host prefix of the form **host-name::** (for example, **beech::/foo/bar**).

In the prefix, **host-name** names the machine running the ObjectStore Server managing the desired ObjectStore directory hierarchy. The example below specifies that a newly created database named **/ken/design/parts1** is to be stored in the hierarchy managed by the Server running on the host named **beech**.

```
os_database *db1=
        os_database::create("beech::/ken/design/parts1");
```

Creating ObjectStore Directories

Note that the directory portion of the path, **/ken/design** in the example above, must be the name of an ObjectStore directory. Unlike databases, which must be created from within a program, ObjectStore directories can be created either from within a program (with the function **objectstore::mkdir()**) or with the ObjectStore utility **osmkdir.** See *ObjectStore Management* for a description of all these options, as well as any platform-specific information that may apply.

Note that pathnames of ObjectStore directories and databases use forward slashes no matter which operating system is being used, and pathnames always begin with a slash (/).

Finding a Rawfs Pathname

The function **os_database::get_pathname()** returns a pathname with a rawfs host prefix.

Aliases

You can use alternative names (aliases) when specifying hosts, but when you use the utility **osls** to determine a database's host machine, the host is identified with its complete, or *canonical*, name. *ObjectStore Management* describes the **osls** utility.

Persistent **new** and **delete**

Most of the time, you create persistent objects with an overloading of **operator new()** provided by ObjectStore. You use **new** just as you would to allocate transient storage, except that you supply a *placement*. Placements are standard in C++. They allow arguments to be supplied to overloadings of **new**.

This section describes the basic use of persistent **new**. More sophisticated uses of persistent **new** allow you to control data *clustering*, and perform transient allocation. See Basic Clustering on page 45.

Creating Nonarrays with Operator new()

Use the following overloadings of **::operator new()** for creating scalar objects in persistent memory:

```
extern void * operator new (
    size_t,
    os_database *where,
    os_typespec *typespec
) ;

extern void * operator new (
    size_t,
    os_segment *where,
    os_typespec *typespec
) ;

extern void * operator new (
    size_t,
    os_object_cluster *where,
    os_typespec *typespec
) ;
```

where specifies *where* in persistent storage the new object is to be stored; that is, you specify the database or, if you would like, the particular segment (object cluster) in which the new object should reside.

typespec specifies the type of the object you are creating. See Using Typespecs on page 36.

The simplest form of **new** requires you to supply just two arguments, an **os_database*** and an **os_typespec***.

```
os_typespec *part_type = new os_typespec("part") ;
part *a_part = new(db1, part_type) part(111) ;
```

In this example, the new object will be stored in the database pointed to by **db1**. Here, as with ordinary **new**, a pointer to the newly created object is returned. All memory allocated by ObjectStore **new** is initialized with zeros.

Creating Arrays with Persistent new()

Use the following overloadings of **::operator new()** for creating arrays of persistent objects:

```
extern void * operator new (
    size_t,
    os_database *where,
    os_typespec *typespec,
    os_int32 how_many
) ;
extern void * operator new (
    size_t,
    os_segment *where,
    os_typespec *typespec,
    os_int32 how_many
) ;
extern void * operator new (
    size_t ,
    os_object_cluster *where,
    os_typespec *typespec,
    os_int32 how_many
) ;
```

how_many specifies the number of elements you want the array to have. For example:

```
part *some_parts = new(db1, part_type, 10) part[10];
```

This call allocates and initializes an array of ten parts. Notice that an **os_typespec** for the type **part** is passed, not **part[]** or **part[10]**. In fact, there are no **os_typespec**s for array types.

OS/2

On OS/2, overloadings of **::operator new[]()** are provided instead of the three overloadings of **::operator new()** above. They have the same arguments and return type.

Persistent Unions

When you define a union type that you intend to have persistent instances, you must supply an associated discriminant function. Discriminant functions are an advanced topic described in

ObjectStore Advanced C++ API User Guide in Chapter 1, Advanced Persistence.

Pointers to and from Persistent Memory

The persistent **new** operator returns a pointer to persistent memory. You use pointers to and from persistent memory in much the same way as you use pointers to and from transient memory. There are a few restrictions that apply, which are discussed in Pointer Validity on page 42.

Clustering

If an object points to an auxiliary data structure, like an associated character array, it is sometimes a good idea to allocate the object and the auxiliary structure within the same *segment*, or region of memory. See Basic Clustering on page 45.

Using Typespecs

Typespecs, instances of the class **os_typespec**, are used as arguments to persistent **new** to help maintain type safety when you are manipulating database roots. A typespec represents a particular type, such as **char**, **part**, or **part***.

For more information about persistent **new**, see Persistent new and delete on page 33, and for more information on type safety see Type Safety for Database Roots on page 51.

Typespecs for Fundamental Types

ObjectStore provides some special functions for retrieving typespecs for types. The first time such a function is called by a particular process, it allocates the typespec and returns 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.

The functions are static members of the class **os_typespec**. Here is a list of their declarations within the definition of **os_typespec**:

```
static os_typespec *get_char();
static os_typespec *get_short();
static os_typespec *get_int();
static os_typespec *get_long();
static os_typespec *get_float();
static os_typespec *get_double();
static os_typespec *get_long_double();
static os_typespec *get_pointer();
static os_typespec *get_signed_char();
static os_typespec *get_signed_short();
static os_typespec *get_signed_int();
static os_typespec *get_signed_long();
static os_typespec *get_unsigned_char();
static os_typespec *get_unsigned_short();
static os_typespec *get_unsigned_int();
static os_typespec *get_unsigned_long();
```

Example: retrieving typespecs

Here is an example:

```
class employee {
    public:
        char *name;
        employee(char* n) {
            name = new(
                os_database::of(this),
                os_typespec::get_char(),
                strlen(n)+1
                ) char[strlen(n)+1];
            strcpy(name, n);
        }
} ;
```

Typespecs for Classes

If you can add a member to a class, the best way to retrieve a typespec for that class is to use a **get_os_typespec()** member function. To do this, add to the class definition the following member function declaration:

```
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.

If you cannot modify the class definition, use the **os_typespec** constructor.

The os_typespec Constructor

It is advisable whenever possible to use typespecs for fundamental types or classes. But if, for example, you could not use typespecs for classes because you are unable to add a new member function to the class definition, you can use the **os_typespec** constructor. In such a case, be sure to follow the guidelines below that illustrate how to limit the activity required in the loop.

To create an **os_typespec**, pass a string (the name of the type you want to create a typespec for) to the **os_typespec** constructor. This works for built-in types (such as **int** and **char**), for classes, and for pointer-to-class and pointer-to-built-in types. The typespec must be allocated in transient memory. Here is an example:

```
os_typespec *part_type = new os_typespec("part");
```

When you create an **os_typespec**, the type-name argument cannot include a space. So, if the type-name argument includes the character ***** (asterisk) for pointer types, the character must *not* be preceded by a space:

```
os_typespec *part_pointer_type = new os_typespec("part*");
```

Once you have created an **os_typespec** for a particular type (the type **part** in the example above), you can use it to create all the program's new instances of that type; there is no need to create a separate **os_typespec** for each call to **new**. Although it is legal, it would be inefficient to do so.

os_typespec models

For example, use the following model:

```
os_typespec part_type ("part");
for (int i=1; i<100000; i++)
    new(db1, &part_type) part (i);
```

Do *not* use the following inefficient model that includes unnecessary repetition within the loop:

```
for (int i=1; i<100000; i++){
    os_typespec part_type("part");
    new(db1, &part_type) part (i);
}
```

Typespecs should only be allocated transiently; you should not create a typespec with persistent **new**.

Parameterized Typespecs

get_os_typespec() member functions (see Typespecs for Classes on page 37) are particularly useful if you are using class templates and you want to create a parameterized typespec. For example, suppose you need a parameterized class that defines a function to persistently allocate an instance of that class and an instance of the parameter. The class might be declared this way:

```
template <class T> class PT {
    . . .
    void foo(os_database *db) {

        /* allocate PT<T> persistently */
        new(db, "PT<???>") PT<T>() ;

        /* allocate type T persistently */
        new(db, "???") T() ;
    }
} ;
```

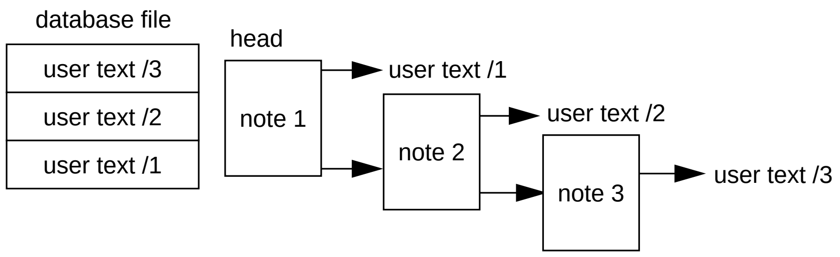
This approach does not work, however, since there is no way to know, when coding, what to fill in for **???**. **T** cannot be used because the C++ template facility will not instantiate it properly — it is inside a string.

The solution is to declare a **get_os_typespec()** member function for the parameterized class, as well as for each class that will serve as parameter.

```
class PT_parm {
    public:
        static os_typespec *get_os_typespec() ;
} ;

template <class T> class PT {

    static os_typespec *get_os_typespec() ;

    void foo(os_database *db) {

        /* allocate new PT<T> persistently */
        new(db, PT<T>::get_os_typespec()) PT<T>() ;

        /* allocate type T persistently */
        /*(assuming it is a suitable class) */
        new(db, T::get_os_typespec()) T() ;
    }

} ;
```

Example: Linked List of Notes

Below is a simple example defining a class that records a note entered by the user. Notes are maintained in reverse order from that in which they were created; that is, the most recent note is at the head of the list. At start-up, the database file is read and the notes are created in memory. The existing notes are displayed and the user is prompted to enter a new note. The database file is rewritten starting with the new note.



Header file: **note.hh**

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

class note {
    public:

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

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

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

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

It is important to recognize that embedding the linked list by means of **note*next;** is not always a reasonable practice. The use of a collection class is a better method of establishing the order of the notes. This method is explained in Chapter 5, Collections, on page 101.

Note program:
main.cc

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

/*Head of linked list of notes */
note* head = 0;
const int note_text_size = 100;

main(int argc, char** argv) {

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

    objectstore::initialize();
    char buff[note_text_size];
    char buff2[note_text_size];
    int  note_priority;
    os_database *db = os_database::open(argv[1], 0, 0644);

    OS_BEGIN_TXN(t1,0,os_transaction::update {

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

        /*  Display existing notes */
        for(note* n=head; n; n=n->next)
            n->display();

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

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

        head = new(db, note::get_os_typespec())
                note(buff, head, note_priority);
        root_head->set_value(head);

    } /* end transaction */
    OS_END_TXN(t1)

    db->close();
}
```

Pointer Validity

Pointers to Persistent Memory

The persistent **new** operator returns a pointer to persistent memory. Use this pointer just as you would use a pointer to transient memory, except for the following:

- Do not use a pointer to persistent memory outside a transaction. If you do, err_no_trans is signaled (except possibly for multithreaded applications — see Threads and Thread Locking on page 84).
- Do not pass a pointer to persistent memory to a non-ObjectStore process, with either a system call or interprocess communication.

The reason for the first restriction concerns concurrency control (see Transactions on page 16) and fault tolerance (see Fault Tolerance on page 70).

The reason for the second restriction concerns ObjectStore’s memory mapping architecture. Pointers to persistent memory are sometimes virtual memory addresses that are, as yet, unmapped. When dereferenced, a hardware fault occurs and ObjectStore handles the fault to retrieve the intended data. If you pass such a pointer to a non-ObjectStore process, and the process dereferences the pointer, ObjectStore will not handle the fault. (See also ObjectStore Memory Mapping Architecture on page 6.)

Cross-Database Pointers

To help boost the performance of certain kinds of applications, ObjectStore’s default mode for new databases and segments does not permit *cross-database*, or external, pointers. In this mode, if a pointer to persistent memory in one database is assigned to persistent memory in a different database, that pointer is valid only until the end of the outermost transaction in which the assignment occurred.

If you want to use cross-database pointers, you can specify particular databases or segments as allowing them. See Referring Across Databases and Transactions on page 61.

Cross-Transaction Pointers

ObjectStore supports a mode in which transient pointers to persistent memory are invalidated at the end of each transaction. In this mode, if you assign to transient memory a pointer to persistent memory, the pointer is valid only until the end of the outermost transaction in which the assignment occurred. This can save on address space consumption and provide a performance improvement for some applications that do not need to retain such pointers across transaction boundaries.

In order to retain such pointers across transaction boundaries you need to understand more advanced operations. See Retaining Pointer Validity Across Transactions in Chapter 1 of *ObjectStore Advanced C++ API User Guide*.

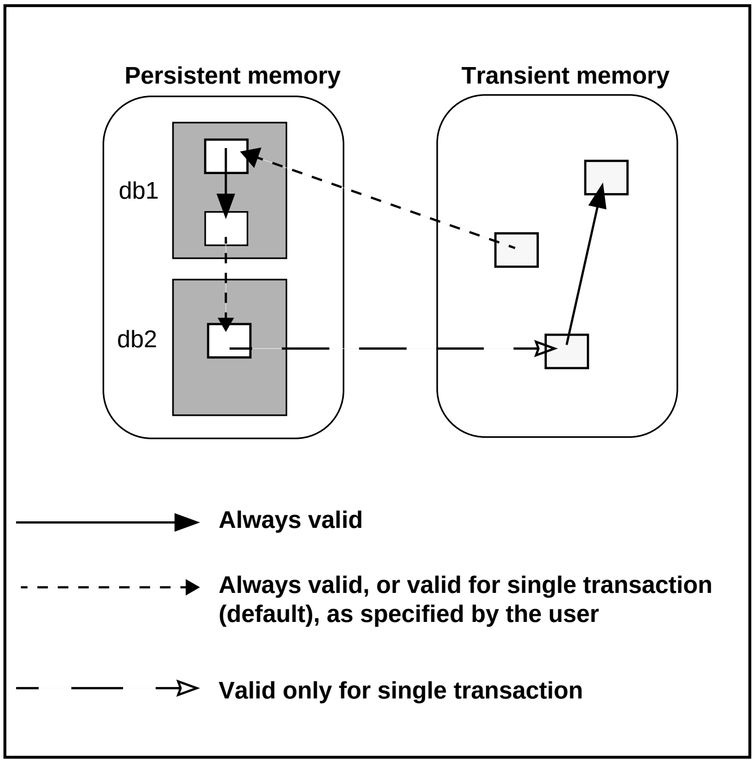
Pointers to Transient Memory

Do not use pointers from persistent to transient memory across transaction boundaries. If a pointer to transient memory is assigned to persistent memory, that pointer is valid only until the end of the outermost transaction in which the assignment occurred.

Use of access hooks

If pointers from persistent to transient memory are useful for your application (for example because some persistent objects point to data structures that you want to take one form in persistent memory and another form in transient memory), you might find ObjectStore *access hooks* useful. See the discussion on **os_database::set_access_hooks()** in the *ObjectStore C++ API Reference*.

Pointer Validity Summary


Persistent memory
Transient memory
db1
db2
Always valid
Always valid, or valid for single transaction (default), as specified by the user
Valid only for single transaction

Basic Clustering

If an object points to an auxiliary data structure, like an associated character array, it is sometimes a good idea to allocate the object and the auxiliary structure in the same database segment. For example, if you want the constructor for the class **employee** to allocate a character array to hold the new employee’s name, you might want to allocate the array in the same segment as the employee.

Segments, as you may recall, are specified regions of memory within a database. The class **os_segment** provides a useful function, **os_segment::of()**, that allows you to determine the *segment* in which a given object resides.

```
os_segment* segment_of() const;
```

Here is how to use **os_segment::of()** to cluster an employee with the employee’s name:

```
extern os_typespec *char_type;

class employee {
    public:
        char *name;
        . . .
        employee(char* n) {
            name = new(os_segment::of(this),
                char_type, strlen(n)+1)
                char[strlen(n)+1];
            strcpy(name, n);
        }
};
```

For information on how to create segments, see Creating Segments on page 60.

Transient Allocation

Writing the constructor in this way has the added advantage that, if the employee is *transiently* allocated, the character array holding the name will be transiently allocated as well. This is because **os_ segment::of()** returns a pointer to the *transient segment* when its argument is transient. Passing the transient segment to ObjectStore’s **new** operator results in transient allocation. This allows you to use ObjectStore **new** to allocate *either* persistent or transient memory, depending on the run-time values of its arguments.

If you specify the transient segment, you can supply **0** for the **os_ typespec*** argument.

Database Entry Points and Data Retrieval

Once you have allocated an object in persistent storage, and stored some data there, how do you retrieve it in subsequent processes? There are four possibilities:

- Look it up by its *persistent name*.
- Retrieve it based on its *persistent name*.
- *Navigate* to it just as you would navigate through transient memory, following pointers.
- Retrieve it using a *query* (see Chapter 5, Queries and Indexes, in the *ObjectStore Advanced C++ API User Guide*).

Database entry point objects

You can look up a persistent object by name if that object is a *database entry point object*, that is, if it has previously been given a persistent name. Establishing Entry Points on page 48 shows you how to name a persistent object, using **os_database_root**, and Retrieving Entry Points on page 50 shows you how to look persistent objects up, using **os_database::find_root()**.

When to Use Persistent Names

It is important to realize that you do not have to give most objects persistent names. This is because you usually retrieve objects in one of the other two ways — either by query or navigation. The only reason you need to name objects is to provide yourself with an *entry point* into persistent memory. Once you have retrieved an entry point object, all objects reachable from it by navigation or query will be automatically retrieved when needed.

For example, suppose that you need to store an assembly in persistent memory. It is typically sufficient to name just the topmost object in the assembly. Once you have retrieved the topmost object using name lookup, you can simply navigate to the other objects in the assembly (assuming each component has a data member pointing to a collection of its children). Since most persistent objects are part of a network of related objects like an assembly, you usually do not have to name or explicitly retrieve them. You only have to do that for one *entry point object*.

Establishing Entry Points

An object can be used as an 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.**

Creating Database Roots

Below are some examples. They show how to associate an object with a name using a root, so the object can be used as an entry point. They also show how to retrieve from the database a pointer to the object by performing a lookup on the name.

```
os_database *db1;
part *a_part;
os_typespec *part_type = new os_typespec("part");
. . .
a_part = new(db1, part_type) part(111);
os_database_root *a_root = db1–>create_root("part_0");
a_root–>set_value(a_part);
```

In this example, a root is created with a member of the class **os_database**, the function **os_database::create_root()**. The function returns a pointer to an instance of the class **os_database_root**. The **this** argument for the function must be the database in which the entry point is stored. If you use the transient database, an err_database_not_open exception is signaled.

This function requires you to specify the name to be associated with the entry point, but the entry point itself is specified in a separate call, using the function **os_database_root::set_value()**.

The return value of **create_root()** is a pointer to the new root. The root is stored in the specified database, in a special segment for database roots. **create_root()** copies the name you supply into this persistent memory, so you can pass a transiently allocated string.

If you pass a name to **create_root()** that is already associated with a root in the specified database, an err_root_exists exception is signaled.

You can get the string associated with a root with the function **os_database_root::get_name()**. This function returns a **char***.

Setting the Value of a Root

The **set_value()** function takes a pointer to the entry point object as argument. It has no return value. The entry point object is first created and stored in the database **db1** through the use of persistent **new**. (This special overloading of **operator new()** is discussed in Persistent new and delete on page 33.)

The pointer you supply as argument to **set_value()** must point to memory in the database containing the root. If it points to transient memory or memory in another database, the exception err_invalid_root_value is signaled.

See also Type Safety for Database Roots on page 51.

Clustering of Roots

You cannot control the clustering of database roots. ObjectStore always stores the roots of each database together in a special segment.

In addition, you can only create a root in persistent storage. If you supply a transient database to **create_root()**, you will get an exception (err_database_not_open).

Retrieving Entry Points

Once one process within an application has created the database root and set its value (to point to the entry point), other processes (or the same process) can retrieve the entry point this way:

```
os_database *db1;
part *a_part;
os_database_root *a_root;
. . .
a_root = db1->find_root("part_0");
if (a_root)
    a_part = (part*) (a_root–>get_value());
```

The **this** argument of **os_database::find_root()** (**db1** in this example) is a pointer to the database containing the root you want to look up. The other argument (**part_0** in this example) is the root's name. If a root with that name exists in the specified database, a pointer to it is returned. If there is no such root, the function returns **0**.

The function **os_database_root::get_value()** returns a **void***, a pointer to the entry point object associated with the specified root. Since the returned value is typed as **void***, a cast is usually required when retrieving it. Here, the returned value is cast to **part***, since the entry point is a **part**.

Use of pvars

You can, alternatively, use ObjectStore *pvars* to maintain valid pointers to an entry point object across transactions. ObjectStore pvars are described in ObjectStore Pvars in Chapter 1 of the *ObjectStore Advanced C++ API User Guide*.

Type Safety for Database Roots

You can gain some additional type safety in your use of database roots by supplying an **os_typespec*** (see Using Typespecs on page 36) as the last argument to **os_database_root::set_value()** and **os_database_root::get_value()**. The typespec should designate the type of the entry point object, the object pointed to by the root's value. The first function stores the typespec in the database, and the second function signals an err_type_mismatch exception if the specified typespec does not match the stored one.

```
os_database *db1;
part *a_part;
os_typespec *part_type = new os_typespec("part");
. . .
a_part = new(db1, part_type) part(111);
db1–>create_root("part_0")–>set_value(a_part, part_type);
```

In this example, notice that the **os_typespec*** argument to **set_value()** and **get_value()** points to a typespec for **part** (not **part***).

Note that **get_value()** checks only that the typespec supplied to it matches the stored typespec, and does not check the type of the entry point object itself.

Deleting Database Roots

If you want to give an object a different name, or stop using it as an entry point, you can delete its associated root:

```
os_database *db1;
. . .
delete db1->find_root("part_0");
```

When you delete a root, the **os_database_root** destructor deletes the associated persistent string as well, but the associated entry point object is not deleted.

Frequently the only way of retrieving an entry point object is through its associated root. So be careful to retrieve such an object *before* deleting its root. Then you can delete it or establish another access path to it.

Application Schemas

As described earlier (see Memory Mapping and Schema Information on page 9), each application and each database has a schema, which consists of the information about classes it uses in a persistent context. You must indicate which classes form an application's schema by supplying one or more *schema source files*, which are used as input to the schema generator (see *ObjectStore Building C++ Interface Applications*).

Including a Class in the Application Schema

The **OS_MARK_SCHEMA_TYPE()** macro

To include a class in the application schema you mark it with a call to the macro **OS_MARK_SCHEMA_TYPE()** inside a function body. You should simply use a dummy function for this purpose (you can name the function anything you want).

You must mark every class on which the application might perform persistent **new**, as well as every class whose instances the application might read from a database. You must also mark every class that appears in a library interface *query string* or *index path* in the application.

Each schema source file must have an include line for each class that it marks. Each schema source file must also include **<ostore/manschem.hh>** after including any ObjectStore header files the application uses. Here is an example:

Example: macro use

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

#include "part.hh"
#include "epart.hh"
#include "mpart.hh"

void dummy() {
    OS_MARK_SCHEMA_TYPE(epart);
    OS_MARK_SCHEMA_TYPE(mpart);
    OS_MARK_SCHEMA_TYPE(part);
}
```

See *ObjectStore Building C++ Interface Applications* for information on class templates and schema source files.

If you supply the **-make_reachable_classes_persistent yes** flag to the schema generator (see *ObjectStore Management*), you do not actually need to mark *every* class used in a persistent context; it is sufficient to mark those classes

- On which the application might perform persistent **new** to create a direct instance of the class
- Whose instances are used by the application as database entry points
- Appearing in a library interface query string or index path

The schema generator performs a *transitive closure operation* on the marked classes to determine the full set of classes reachable from the specified classes. This way, other classes used in a persistent context, such as a class of object embedded in or pointed to by an entry point object, are included in the application schema.

For applications that link with libraries that operate on persistent data, the application schema includes the schemas for the libraries. Library schemas are recorded in library schema databases (see *ObjectStore Building C++ Interface Applications*), and are themselves generated from schema source files.

Database Schemas

As described earlier (see Memory Mapping and Schema Information on page 9), each application has a schema, which consists of the classes it uses in a persistent context. Each database has a schema as well. This section describes when and how classes are added to database schemas, and explains when an application schema and a database schema are *compatible*.

A database’s schema is determined by the schemas of the applications that access the database. An application augments the schema of a database in one of two ways: through *batch schema installation* or through *incremental schema installation*. Batch installation is the default.

Batch Database Schema Installation

With batch schema installation, whenever an application first accesses a database, each class in the application’s schema that might be persistently allocated is added to the database’s schema (if not already present in the database schema). Subsequent runs of the application will not have to install schema in that database, unless the application’s schema changes (as evidenced by a change in date of the application schema database).

Incremental Database Schema Installation

If you want you can specify, for a particular database, that schema installation should be incremental. 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.

Incremental schema installation by database

By default, each new database is in batch installation mode. You can change this by using the **OS_INC_SCHEMA_INSTALLATION** environment variable, described in *ObjectStore Management*, or by using the **os_database::set_incremental_schema_installation()** function. Performing this function on a database, with a nonzero 32-bit integer (true) as argument, causes the applications that subsequently open the database to install schema in an incremental fashion. The function is declared

```
void os_database::set_incremental_schema_installation(
        os_boolean
) ;
```

os_boolean is a 32-bit integer type defined as **int** or **long**, depending on platform. Calling this function with **0** (false) as its argument sets the specified database's installation mode to batch.

Incremental schema installation by application

Calling the static member function **objectstore::set_incremental_ schema_installation()** with a nonzero integer (true) as argument specifies that the current application run should perform incremental schema installation on all databases it accesses, regardless of the database's mode.

This function also specifies the schema installation mode for databases yet to be created by an application. If an application calls this function with a nonzero integer (true) as argument, 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. This function is declared

```
static void objectstore::set_incremental_schema_installation(
    os_boolean
) ;
```

Comparison Between Batch and Incremental Installation

Advantages of incremental installation

Incremental installation has the following advantages over batch installation:

- Creating an empty database is faster, since no user schema is installed at the time of creation (a minimal internally used schema is installed at create time).
- Database schemas will in general be smaller, since only the classes actually required to describe objects in the database will be installed.
- Since database schemas will not contain unnecessary classes, spurious incompatibilities between application and database schemas will be reduced.

Disadvantages of incremental installation

The disadvantages of incremental installation, as compared with batch installation, are as follows:

- The first persistent allocation of an object of a given class will be slower, since it involves installation of that class into the database schema. (The speed of subsequent allocations of objects of that class is unchanged with regard to batch schema installation.)

- If an application actually uses all the classes in its schema for access to a given database, the total time to install the schema will increase, because of the overhead associated with incremental schema merging.

Timing of Installation

Batch schema installation occurs when a database is created, assuming neither the database nor the application is in incremental installation mode. For databases not created by the application, it occurs the first time the application accesses the database, within an update transaction, while it is opened for read / write — again, assuming neither the database nor the application is in incremental installation mode. Note that modifying the application’s schema is considered to result in a different application.

Schema Validation

When an application accesses a database, the application's schema and the database's schema must be compatible. ObjectStore checks for this compatibility during *schema validation*. For each transaction in which an application accesses a database, ObjectStore checks to make sure that the definitions of all classes in both schemas agree. In other words, if a class named **C** appears in both the application schema and the database schema, ObjectStore verifies that the application schema's definition of **C** agrees with the database schema's definition of **C**. Two class definitions agree, in this sense, if and only if

- They have the same base classes in the same order.
- They have the same data members declared in the same order and with the same value types.
- Either they both define no virtual functions or they both define at least one virtual function.
- They have the same set of discriminant function names (see Discriminant Functions, discussed in Chapter 1 of the *ObjectStore Advanced C++ API User Guide*).

If an application and database schema are incompatible, the exception err_schema_validation_error is signaled.

Specific Schema Verification Implementation

Schema verification is implemented to be strictly based on assignment compatibility and size. That is, if C++ defined a compatible assignment of a value of type **T1** to a location of type **T2** and

sizeof(T1) == sizeof(T2)

then and only then would **T1** and **T2** verify validly. Consequently,

1. Any type **T** is compatible with any other type **T**.
2. **char**, **unsigned char**, and **signed char** are compatible.
3. **short**, **unsigned short**, and **signed short** are compatible.
4. **int**, **unsigned int**, and **signed int** are compatible.
5. **long**, **unsigned long**, and **signed long** are compatible.

6 An array **T1[***n1***]** is only compatible with **T2[***n2***]**.

T1 == T2

and

n1 **==** *n2*

(Strictly speaking, **short**, **int**, and **long** are the same as **signed short**, **signed int**, and **signed long** respectively.)

With pointers, however, C++ assignment compatibility is relaxed when the referent is an integral type. Pointer **T1*** is compatible with pointer **T2*** if

T1 == T2

or **T1** and **T2** are covered by rules 2, 3, 4, or 5 above. That is, **T1** and **T2** are integral types of the same size.

Timing of Validation

Schema validation occurs for a database the first time in each transaction that the application accesses the database. Note that implicit database access sometimes occurs after a transaction has been initiated but before execution of the transaction's first statement. For this reason, if you want to handle err_schema_validation_error, the handlers should generally enclose transactions in which database access might occur.

Creating Segments

Every database when first created contains a default segment and, if its schema is not stored remotely, a *schema segment*. (The schema segment contains schema information used internally by ObjectStore, as well as all the database's roots.)

If you use the simple form of persistent **new** described in Persistent new and delete on page 33 (specifying only a database, not a segment), the new object is stored in the default segment. This segment, like all segments, expands to accommodate whatever new data is stored in it. If you want a database to have a new segment, you create it using the member function **os_ database::create_segment()**.

```
os_segment *create_segment() ;
```

The function **os_database::create_segment()** takes as **this** argument a pointer to the database in which the segment is to be created. This function takes no other arguments. It returns a pointer to an instance of the system-supplied class **os_segment**.

As with instances of **os_database**, the segment object is actually *transient*, unlike the segment it stands for. That is, it exists only for the duration of the current process. If you copy it into persistent storage, it will be meaningless when retrieved by another process.

See Basic Clustering on page 45, as well as Creating Object Clusters in Chapter 1 of the *ObjectStore Advanced C++ API User Guide*.

Referring Across Databases and Transactions

If you want to use cross-database pointers, you can specify particular databases or segments as allowing such pointers. This is done with the function **os_database::allow_external_pointers()** or **os_segment::allow_external_pointers()**. These functions must be called from within a transaction.

allow_external_ pointers()

```
void allow_external_pointers() ;
```

Here is an example of a call to the **os_database::allow_external_ pointers()** function.

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

void f() {

    objectstore::initialize();
    static os_database *db1 = os_database::open("/thx/parts");

    OS_BEGIN_TXN(tx1, 0, os_transaction::update)

        db1->allow_external_pointers();

        /* now cross-database pointers from db1 are allowed */
        . . .

    OS_END_TXN(tx1)

    db1–>close();
}
```

Once you perform **allow_external_pointers()** on a database, the current process and subsequent processes can store cross-database pointers there. When you access a cross-database pointer, if the database it points to is not open, it is opened automatically.

There is a small performance cost to allowing cross-database pointers. Allowing external pointers augments the entries in the tables that associate virtual memory addresses with on-disk object locations. This slightly increases the time it takes for ObjectStore to transfer data into memory.

This increase applies to all data stored in the database from which external pointers are allowed. You can localize the performance cost by allowing external pointers from a particular segment or segments rather than from the entire database.

Example

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

void f() {

    objectstore::initialize();

    static os_database *db1 = os_database::open("/thx/parts");
    static os_segment *seg1 = db1->get_default_segment();

    OS_BEGIN_TXN(tx1, 0, os_transaction::update)

        seg1->allow_external_pointers();

        /* now cross-database pointers from seg1 are allowed */
        . . .

    OS_END_TXN(tx1)

    db1–>close();
}
```

In addition to localizing the performance cost, allowing cross-database pointers on a per-segment basis gives you greater control over integrity management, since what counts as an illegal pointer can be different from segment to segment within the same database.

Cross-database pointers are resolved according to the *relative pathname* of the database containing the referenced data. See Cross-Database Pointers and Relative Pathnames on page 63.

Cross-transaction pointers

You can also allow cross-transaction pointers in the current process by calling the static function **objectstore::retain_ persistent_addresses()**. See Retaining Pointer Validity Across Transactions in Chapter 1 of the *ObjectStore Advanced C++ API User Guide*.

ObjectStore references

If you only need to refer across databases in a few cases, you can avoid the slight performance cost of allowing cross-database pointers from an entire database or segment by using ObjectStore references instead. You can also use ObjectStore references to refer across transaction boundaries on a case-by-case basis, instead of using **retain_persistent_addresses()**. See Using ObjectStore References in Chapter 1 of the *ObjectStore Advanced C++ API User Guide*.

Cross-Database Pointers and Relative Pathnames

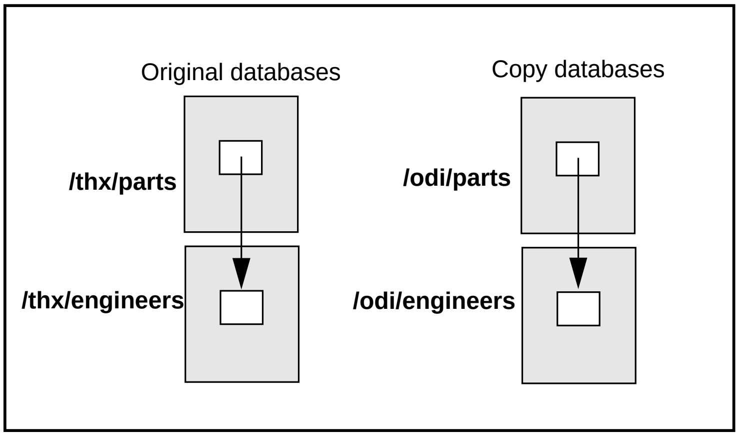
Cross-database pointers are resolved according to the *relative pathname* of the database containing the referenced data. For example, suppose a database with pathname **/thx/parts** contains a pointer to data in a database with pathname **/thx/engineers**. Since these two databases have a common ancestor directory (**/thx**), pointers to **/thx/engineers** are resolved according to the relative pathname **../engineers**. (Remember that pathnames of ObjectStore directories and databases use slashes no matter which operating system is being used, and pathnames always begin with a slash (**/**).)

In implementation terms, this means that ObjectStore records only a relative pathname when recording the on-disk location of the referent object (in the tables that associate virtual memory addresses with on-disk locations; see ObjectStore Memory Mapping Architecture on page 6).

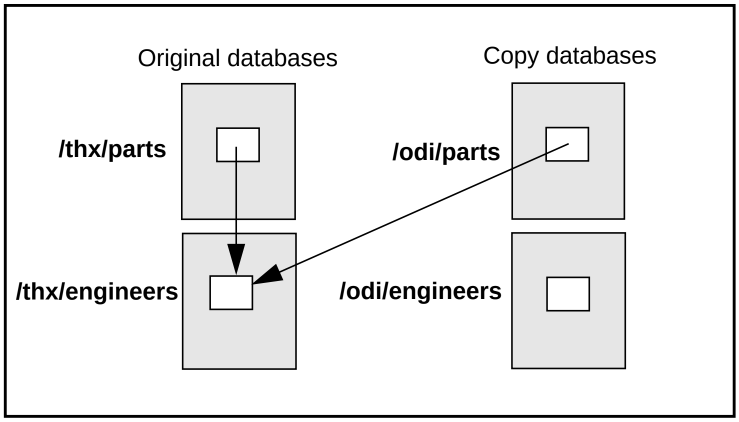
In practical terms, resolution by relative pathname can be understood as follows. Suppose that you copy the two databases **/thx/parts** and **/thx/engineers**, giving the copies the pathnames **/odi/parts** and **/odi/engineers**. Because resolution is by relative pathname, the copy of the outgoing pointer from the parts database will refer to data in the copy of the engineers database, **/odi/engineers**. If the cross-database pointer had been resolved according to the *absolute* pathname of the referent database, the copy of the pointer would have referred to data in the original database, **/thx/engineers**.

The following diagrams show the difference between resolution according to relative and absolute pathnames.

Resolution of cross-database pointers by relative pathname



Resolution of cross-database pointers by absolute pathname



Default Relative Directory

The pathname used in the default resolution procedure is relative to the *lowest* directory in the hierarchy that the two pathnames have in common. For example, if a pointer is stored in **/A/B/C/db1** that refers to data in **/A/B/D/db2**, the lowest common directory is **A/B**, so the relative pathname **../../D/db2** is used to record the referent data's on-disk location.

Specifying a Relative Directory

If you want, you can explicitly specify the relative directory by using the function **os_database::set_relative_directory()**. To see how this works, consider the previous example: you want to store a pointer in **/A/B/C/db1** that refers to data in **/A/B/D/db2**. Suppose that you want the pointer resolved according to the relative pathname **../B/D/db2**; that is, you want the relative directory to be **/A** (not **/A/B** as it would be in the default case).

To ensure this, before you store the pointer, you set the relative directory like this:

Example: setting the relative directory

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

void f() {

    objectstore::initialize();

    static os_database *db1 = os_database::open("/thx/parts");
    static os_segment *seg1 = db1->get_default_segment();

    OS_BEGIN_TXN(tx1, 0, os_transaction::update)
        seg1->allow_external_pointers();
        db1->set_relative_directory("/A"));
        . . .
    OS_END_TXN(tx1)

    db1–>close();
}
```

Then, for the rest of the process, cross-database pointers will be stored using pathnames that are relative to the directory you specified (**/A** in the example). When a process sets the relative directory, the effect is local to that process. Other processes, including concurrent ones, can set a different relative directory or use the default behavior.

If you specify null (**0**) instead of a string as the argument to **os_ segment::set_relative_directory()**, the default procedure is used to determine the relative directory.

Using Absolute Pathnames

If you want cross-database pointers stored with absolute pathnames, you should specify the empty string as the argument to **os_segment::set_relative_directory()**.

The oschangedbref Utility

To change the database a pointer refers to, you can use the ObjectStore utility **oschangedbref**. See *ObjectStore Management* for more information.

Ensuring Data Access During System Calls

ObjectStore's Virtual Memory Mapping Architecture normally ensures that as you access persistent locations, the necessary pages of virtual memory are made available to be read or written. It does this by intercepting memory access violation faults, mapping the page or pages, and continuing the faulting instruction.

However, ObjectStore's fault handler is circumvented for cases in which you pass a persistent pointer to one of the following:

- A system call such as **read** or **ReadFile**
- A library function such as **fread** that might end up using such a system call
- A function that itself traps faulting memory references (such as **lstrlen** on Windows)

Instead, the system call returns some sort of error indication (usually EFAULT on UNIX, ERROR_INVALID_PARAMETER on Windows, or a **0** return from functions such as **lstrlen**).

To ensure that data is accessible during a system call, create an automatic **os_with_mapped** object, specifying the starting address and size of the range you want to pass to a system call, and whether you intend to update the object.

Within the scope of the **os_with_mapped** object, the referenced range is available to the system call. For example:

```
{
  os_with_mapped mybuf(persistent_buffer, buffer_size, 1);
  read(fd, persistent_buffer, buffer_size);
}
```

The constructor for **os_with_mapped** ensures that the necessary pages are available and mapped with appropriate access rights, and that those pages are wired into the client cache until the destructor is run. The constructor signals an exception if you run out of room in the cache, if you attempt to wire a page more than 250 times, or if obtaining a page fails because of deadlock.

The destructor allows the pages to be moved out of the cache again as necessary.

Object Design recommends the use of **os_with_mapped** whenever you pass a persistent pointer to system calls or library functions that might call system calls or handle memory access violations. There is little overhead to **os_with_mapped**, so calling it frequently does not generally present a performance problem.

Mapping large ranges, having many simultaneous mappings, or keeping mappings in effect for a long time is not recommended, because this can interfere with cache replacement and lead to a cache-is-full exception.

Restriction

An **os_with_mapped** object cannot be used across transaction boundaries, including nested transactions.

Chapter 3
Transactions

The information about transactions is organized in the following manner:

Transactions Overview

A transaction is a logical unit of work, a consistent and reliable portion of the execution of a program. You mark the beginnings and ends of transactions in your code using calls to the ObjectStore API. Access to persistent data must always take place within a transaction.

Transactions in a database system serve two general purposes:

- They support fault tolerance.
- They support concurrent database access.

Fault Tolerance

In support of fault tolerance, transactions have the following properties:

- Either all 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 are made.
- 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.

Fault tolerance is implemented using a *transaction log*. For further details, see Logging and Propagation in Chapter 2 of the *ObjectStore Advanced C++ API User Guide*.

Concurrency Control

Transactions support concurrent database access by preventing one process’s updates from interfering with another process’s reads or updates. ObjectStore’s concurrency control facilities prevent this interference by ensuring that transactions have the following properties:

- A transaction’s changes to persistent data are private and invisible to other processes until the entire transaction completes successfully.
- Other processes’ changes to persistent data are invisible to a transaction.

Concurrency control is implemented using strict two-phase locking (see Locking on page 77), and also — in the case of abort-only transactions and *multiversion concurrency control* (MVCC) — using special techniques of delaying propagation and intentionally aborting transactions. See Using Dynamic Transactions on page 76, as well as the discussion on Multiversion Concurrency Control (MVCC) in Chapter 2 of the *ObjectStore Advanced C++ API User Guide*.

Transaction Commit and Abort

Transactions can terminate in two ways: successfully or unsuccessfully. When they terminate successfully, they *commit*, and their changes to persistent memory are made permanent and visible. When they terminate unsuccessfully, they *abort*. There are several kinds of transaction aborts:

- Explicit aborts, which result from calls to transaction member functions. See Rolling Back to Persistent State on page 81.
- Aborts due to deadlock conditions. See Threads and Thread Locking on page 84.
- Aborts due to nonlocal transfer of control out of the scope of a *lexical transaction*. (See Using Transactions on page 72.) This can happen when an exception is signaled within a lexical transaction and handled outside it, or not handled at all.
- Aborts due to system failure.

If a transaction aborts, its changes to persistent memory are *not* made permanent or visible to other processes. After an abort, your program sees persistent memory as it was just before the aborted transaction started. But only persistent memory changes are rolled back. Transient memory is not restored to its pretransaction state however, and any form of output that occurred before the abort is not, of course, undone.

Using Transactions

With ObjectStore, every statement that reads from or writes to persistently allocated memory must be within a *transaction*. If you attempt to access persistent data outside a transaction, err_no_trans is signaled.

This applies to statements that access data *in* a database, but not to all statements that operate *on* a database. Statements that create, open, or close a database can be either inside or outside a transaction, although, generally, it is advisable not to open or close a database within a transaction.

Lexical and Dynamic Transactions

There are two ways to mark off transactions with ObjectStore:

- *Lexical transactions* require that the transaction be contained in a lexical context. You mark off lexical transactions with the ObjectStore transaction macros. See Using Lexical Transactions on page 74.
- *Dynamic transactions* provide dynamically defined boundaries. You mark off dynamic transactions with members of the class **os_transaction**. See Using Dynamic Transactions on page 76.

Some applications require transactions with dynamically defined boundaries. A typical scenario for using dynamic transactions is in loops that commit periodically. For these programs, use dynamic transactions.

Choosing Transaction Boundaries

When you mark off transactions in your code you must balance the following two considerations:

- If a transaction includes too much, it can negatively affect the performance of other concurrent applications.
- If a transaction includes too little, other concurrent applications can interfere with the application containing the transaction, causing the application to produce incorrect results. Additionally, a greater number of transactions can increase overhead.

To help you determine how to demarcate your transaction boundaries, look at Locking on page 77 as well as Organizing Transaction Code on page 79.

Multiversion Concurrency Control (MVCC)

When you use *multiversion concurrency control* (*MVCC*), you can perform nonblocking reads of a database, allowing another ObjectStore application to update the database concurrently, with no waiting by either the reader or the writer. If your application contains a transaction that uses a database in a read-only fashion, you might be able to use multiversion concurrency control. See Multiversion Concurrency Control (MVCC) in Chapter 2 of the *ObjectStore Advanced C++ API User Guide* for more information.

Using Lexical Transactions

You begin and commit lexical transactions with the following macros:

OS_BEGIN_TXN(*identifier,exception**,transaction-type***)**

and

OS_END_TXN(*identifier***)**

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 you must enter these macro arguments *without white space* to ensure that the argument concatenation will work correctly.

These and other ObjectStore macros are described in Chapter 4, System-Supplied Macros, in the *ObjectStore C++ API Reference*.

identifier is a transaction tag. The only requirement on the tag is that different transactions in the same function must use different tags. (The tags are used to construct statement labels, and so have the same scope as labels in C++.)

*exception*** specifies a location in which ObjectStore will store an *exception** if the transaction is aborted because of the raising of an exception. Raising an exception will cause a lexical transaction to abort if the exception is handled outside the transaction's dynamic scope, or if there is no handler for the exception. The stored *exception** indicates the exception that caused the abort. ObjectStore stores **0** in this location at the beginning of each transaction.

Transaction type enumerators

transaction-type is one of the following enumerators, defined in the scope of **os_transaction**:

- **os_transaction::update** specifies a transaction in which updates to persistent memory are allowed.
- **os_transaction::read_only** specifies a transaction in which any attempt to update persistent memory signals the exception err_write_permission_denied.
- **os_transaction::abort_only** specifies a transaction in which writes to persistent memory are allowed, but the transaction cannot be committed.

If a lexical transaction is aborted due to deadlock, it is automatically retried. See Threads and Thread Locking on page 84.

Example: a lexical transaction

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

main(int, char **argv) {

    os_database *db1 = os_database::open( argv[1] ) ;

    OS_BEGIN_TXN(my_tx_1,0,os_transaction::update)

        int countp* = (int*)( db1->find_root("count")->get_value() ) ;
        cout << "Hello, world\n" ;
        cout << ++*countp << "\n" ;

    OS_END_TXN(my_tx_1)

    db1–>close() ;
}
```

Using Dynamic Transactions

You start and commit dynamic transactions with the following members of the class **os_transaction**:

```
static os_transaction *begin(
    os_int32 transaction_type = os_transaction::update
) ;
static void commit() ;
static void commpwd
it( os_transaction* ) ;
```

The statements executed in between the calls are all within the same transaction.

Transaction type enumerators

transaction_type is one of the following enumerators, defined in the scope of **os_transaction**:

- **os_transaction::update** specifies a transaction in which updates to persistent memory are allowed.
- **os_transaction::read_only** specifies a transaction in which any attempt to update persistent memory signals the exception err_write_permission_denied.
- **os_transaction::abort_only** specifies a transaction in which writes to persistent memory are allowed, but the transaction cannot be committed. An attempt to commit the transaction signals the exception err_commit_abort_only.

begin() returns a pointer to a transaction, an instance of the class **os_transaction**.

The first overloading of **commit()** commits the current transaction. In the case of nesting, it commits the most nested transaction. The second overloading of **commit()** commits the specified transaction.

Unlike lexical transactions, if a dynamic transaction is aborted due to deadlock, it is not automatically retried. See Threads and Thread Locking on page 84.

Locking

As with most database systems, ObjectStore tries to interleave the operations of different processes' transactions to maximize concurrent usage of resources. When scheduling the operations, ObjectStore conforms to the strict two-phase locking discipline (except in the case of *multiversion concurrency control* as described in Multiversion Concurrency Control (MVCC) in Chapter 2 of the *ObjectStore Advanced C++ API User Guide*). This discipline has been proven correct in the sense that it guarantees serializability; that is, it guarantees that the results of the schedule will be just the same as the results of noninterleaved scheduling of the transactions' operations.

Waiting for Locks

Roughly speaking, when you access data in the database, you are given exclusive access to that data for the *duration* of the transaction in which the access takes place. That is, when you access data, that data is *locked*. As long as it is locked, no other process can access it. The data is not *unlocked* until the end of the transaction.

Database- Compared to Segment-Level Locks

There are different kinds of locking provided by database and segment level locks. As its name implies, a *database lock* prohibits access to the entire database. A *segment-level lock* only blocks access to the specific segment affected by the transaction.

Read Locks and Write Locks

Locking actually treats reading data differently from writing data. When your process *reads* a persistent data item (such as a data member or persistent variable), the page on which the item resides is *read locked*. This prevents other processes from writing to that page, but they are still allowed read access to it. When your process *writes* a data item, the page on which it resides is *write locked* unless the transaction is **abort_only**. If the transaction is **abort_only**, the client obtains read locks for all pages read or written but does not get any write locks. This prevents other processes from reading or writing to that page. (See Transaction Locking Examples in Chapter 2 of the *ObjectStore Advanced C++*

API User Guide, as well as **os_transaction::abort_only** in the *ObjectStore C++ API Reference*.)

Lock Timeouts

You can set a timeout for read- or write-lock attempts, to limit the amount of time your application will wait to acquire a lock. When the timeout is exceeded, an exception is signaled. Handling the exception allows you to continue with alternative processing, and make a later attempt to acquire the lock. See the **set_readlock_ timeout()** and **set_writelock_timeout()** members of the classes **objectstore**, **os_database**, and **os_segment** in the *ObjectStore C++ API Reference*.

Reducing Wait Time

There are a number of ways to minimize the amount of time your process spends waiting for locks. See Reducing Wait Time for Locks in Chapter 2 of the *ObjectStore Advanced C++ API User Guide*.

Lock Probes

You can determine whether a specified address is read locked, write locked, or unlocked with **objectstore::get_lock_status()**. See the *ObjectStore C++ API Reference*.

Explicit Lock Acquisition

Normally, ObjectStore performs locking automatically and transparently to the user. But you can explicitly lock a specified page range for read or write with **objectstore::acquire_lock()**. See the *ObjectStore C++ API Reference*.

Organizing Transaction Code

If you make transactions too short, you might be allowing other processes to interfere in a harmful way with your process. That is, if some chunk of your code is grouped into two or more short transactions when it should really be all within a single longer transaction, your process or others could produce incorrect results. Here are some guidelines about how to organize your code into transactions.

Guidelines for organizing code within a transaction

In general, you should put a given chunk of code inside a single transaction when

- You do not want other processes to see intermediate results of this code's execution.
- You want the state of the database to be frozen, from the point of view of the code being executed, for the duration of its execution. That is, no changes to the database made by other concurrent processes should be visible for the duration of the code's execution.

Another reason to put a chunk of code in a single transaction is to allow you to undo the code's changes at any point before the end of the chunk. See Rolling Back to Persistent State on page 81.

Hiding Intermediate Results

One kind of interference between processes occurs when one process uses some *intermediate results* of another. Just what constitutes an intermediate result depends on the application. Consider, for example, an imaginary MCAD application.

Suppose each of two processes is replacing two different children of a given part. Suppose further that each process must make some constraint check on the assembly after the replacement has been performed. Perhaps the total cost of the assembly must be checked against some allowable maximum cost.

In replacing a subpart, the first process removes a child part from the set of the assembly's children, and then inserts a different part into this set. But between the remove and insert, the assembly is in an intermediate state that should not be visible to the other process. Suppose, for example, the second process does its part replacement while the assembly is in this intermediate state, and then performs a cost check. The cost will be incorrect (too low), since a subpart is missing from the assembly. If the second process's new part raises the actual cost above the maximum, this will go undetected.

To prevent exposure of such intermediate states, the process should put the remove and the insert into the same transaction. This way, as far as other processes are concerned, the replacement happens *all at once*. In general, whatever happens within a single transaction looks to other processes as if it happens instantaneously, since the intermediate states are not visible to them.

Preventing Other Processes' Changes

Another kind of interference between processes arises when one process relies on the state of persistent memory's being unaffected by other processes for the duration of some operation.

Consider, for example, a routine that involves a recursive descent of a given assembly. Suppose that another process removes a subpart from the assembly, but it does not matter whether the descent is performed before or after the removal. Nevertheless, for this process to produce correct results, the assembly's descendents must not change during the descent itself. For if a subpart is removed after being visited, and then, before this removed subpart's children are visited, new children are added to it, these new children might be incorrectly visited as part of the original assembly's descendents. So all the code that performs the descent should be within the same transaction.

Rolling Back to Persistent State

If a transaction aborts, its changes to persistent memory are not made permanent or visible to other processes. After an abort, your program sees persistent memory as it was just before the aborted transaction started. You can abort a specified transaction using members of the class **os_transaction**. You can also abort a lexical transaction by signaling an exception within the transaction and handling the exception outside the transaction.

Aborting the Current Transaction

You can always roll back to the persistent memory state at the beginning of the current transaction (the most deeply nested transaction within which control currently resides) by calling the following member of the class **os_transaction**:

```
static void abort() ;
```

For dynamic transactions, control flows to the next statement that follows the **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.

Aborting the Top-Level Transaction

When you call **os_transaction::abort()** with no arguments, only the innermost transaction is aborted. But you can abort the outermost transaction with a call to the static member function **os_ transaction::abort_top_level()**, with no arguments.

```
static void abort_top_level() ;
```

Aborting a Specified Transaction

You can also specify a transaction in between, by including an argument in an **os_transaction::abort()** call.

```
static void abort(os_transaction*) ;
```

The argument is a pointer to a transaction, an instance of the system-supplied class **os_transaction**. A pointer to the current transaction (the innermost transaction in which control currently resides) is returned by the static member function **os_ transaction::get_current()**.

```
static os_transaction *get_current() ;
```

A pointer to its parent (the innermost transaction within which it is nested) is returned by the member function **get_parent()**.

```
os_transaction *get_parent() const ;
```

So, for example, to abort a transaction one level up from the current transaction, you might use the following code:

```
os_transaction* child_tx = os_transaction::get_current() ;
if (child_tx) {
    parent_tx = child_tx->get_parent() ;
    if (parent_tx)
        os_transaction::abort(parent_tx) ;
}
```

Example: **abort()**

Consider an example involving replacement of an assembly’s subparts. A constraint check is required after each replacement. If the constraint check fails, you would like the replacement to be undone. To do so, you can conditionally call **os_ transaction::abort()**, as in the code below:

```
main() {

    os_database *db5 = os_database::open("/user1/db5");

    OS_BEGIN_TXN(tx1,0,os_transaction::update)

        os_typespec *part_type = ...;
        part *a_wheel = ...;
        part *a_rim = ...;

        a_wheel–>children -= a_rim;
        /* in this intermediate state, the wheel has no rim */
        /* but this state is not visible to other processes */
        a_wheel–>children |= new(db5, part_type) part(...);

        if (!check_cost(a_wheel)) {
            cout << "change aborted: cost check failed\n";
            /* undo the part replacement* /
            os_transaction::abort();
        } /* end if */

    OS_END_TXN(tx1)
    db5–>close();
}
```

Since the abort results in control's leaving the scope of the current transaction, the current state of all local transient memory is lost. But transient state that is not local to this scope is unaffected by the abort. You should explicitly roll back or reinitialize such state before the abort, if desired.

Threads and Thread Locking

If your application uses multiple threads, you might need to take advantage of the thread-locking facilities provided by ObjectStore. These facilities ensure that ObjectStore does all interlocking between threads necessary to prevent threads from interfering with one another when within the ObjectStore run time. You are responsible for coding any thread synchronization required by your application while threads are not executing within an ObjectStore library. See Chapter 3, Threads, in the *ObjectStore Advanced C++ API User Guide* for more information about thread locking in multithreaded applications.

The thread-locking facility works by either serializing the transactions of different threads or serializing access by different threads to the ObjectStore run time. No two threads are ever in the ObjectStore run time at the same time.

Thread Safety

ObjectStore supports thread safety using a global *mutex*. This is a data structure that is used to synchronize threads. One global mutex coordinates all threads within an application. Thus, access to the ObjectStore API is currently serialized with one global mutex.

ObjectStore Release 5.1 provides a thread-safe version of the ObjectStore API. It does this by protecting the body of each API call with a mutex lock that only one thread can acquire at a time.

When You Need Thread Locking

If 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. A thread can enter the ObjectStore run time under either of the following circumstances:

- The thread dereferences a pointer to persistent memory.
- The thread calls an ObjectStore API function or macro.

If only one thread at a time ever enters the ObjectStore run time, you should disable ObjectStore thread locking. Do not use thread locking if you do not have to, since there is some extra performance overhead associated with it.

Disabling and Enabling Thread Locking

ObjectStore thread locking is enabled by default. To enable ObjectStore thread locking explicitly, pass a nonzero value to the following member of the class **objectstore**:

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

To disable ObjectStore thread locking, pass **0** to this function. To determine if ObjectStore thread locking is enabled, use the following member of **objectstore**:

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

If nonzero is returned, ObjectStore thread locking is enabled; if **0** is returned, ObjectStore thread locking is disabled.

Local and Global Transactions

For applications that use multiple threads, there are two kinds of transactions: local transactions and global transactions. Transactions started with **OS_BEGIN_TXN()** are always local. Transactions started with **os_transaction::begin()** are local by default, but you can also request a global dynamic transaction. See Using Global Transactions on page 86.

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 by the mutex lock 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. See Chapter 3, Threads, in the *ObjectStore Advanced C++ API User Guide* for more information on using threads with ObjectStore.

Costs and Benefits of Global Transactions

Advantages of global transactions

Local transactions usually provide better performance, but for some applications, global transactions might be preferable. Here are some of the benefits of using global transactions:

- Global transactions allow for a higher degree of concurrency.
- With local transactions, if one thread attempts to access persistent memory from outside a transaction while another thread is performing relocation, data corruption can result. No exception is signaled. With global transactions (as in the absence of threads), there is no such possibility. Any attempt to access persistent memory from outside a transaction results in err_no_trans.

Disadvantages of global transactions

Some of the disadvantages of using global transactions are

- Global transactions have extra overhead, compared to local transactions, in the form of extra memory management, particularly if there is a lot of cache replacement.
- With global transactions, you must synchronize the threads so that no thread attempts to access persistent data while another thread is committing or aborting.

Using Global Transactions

You start a global transaction by passing the enumerator **os_transaction::global** as the second argument to **os_transaction::begin()**.

```
enum os_transaction_scope {
    os_transaction::local = 1,os_transaction::global
};

static os_transaction::begin(
        os_int32 type = os_transaction::update,
        os_int32 scope = os_transaction::local
    );
```

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. If you do not, data corruption and program failure can result.

The exception err_deadlock might be signaled asynchronously in any thread using persistent data; the application must be prepared to handle it. Once err_deadlock is handled in the first thread, any other threads that attempt to use the transaction will also get err_deadlock; in particular, any threads that were waiting for the global lock will wake up and immediately get err_deadlock.

Nesting and Global Transactions

You cannot nest a local transaction within a global transaction, nor can you nest a global transaction within a local one. The following table specifies how two transactions can interact.

	Thread A runs global transaction	***Thread A runs local transaction***
Thread A tries global transaction	OK. Nested global transaction.	err_trans_wrong_type is signaled.
Thread A tries local transaction	err_trans_wrong_type is signaled.	OK. Nested local transaction.
Thread B tries global transaction	OK. Nested global transaction.	OK, but block until A completes.
Thread B tries local transaction	err_trans_wrong_type is signaled	OK, but block until A completes.

Additional information about nested transactions is in Chapter 2, Advanced Transactions, of the *ObjectStore Advanced C++ API User Guide*. For further discussion of threads, see Chapter 3, Threads, of that publication.

Chapter 4
Notification

The information about notification is organized in the following manner:

Notification Overview

The notification service allows an ObjectStore client to notify other ObjectStore clients that an event has taken place. Typically, a notification event corresponds to the modification of an object in a database, but applications are free to assign their own meanings to events.

A notification broadcasts to *subscribers* that an event (for example, a change) has occurred at a database location (for example, the location of a persistent object). ObjectStore applications can subscribe to receive notifications that are posted on a database location, or on a range of database locations. If a range is specified in a subscription, notifications posted on any location in the range are received by the subscribing application. Subscribers can poll for notifications, or *block* (remain in a wait state) until a notification is posted.

When an application posts a notification, it specifies the database location, an integer code, and a character string. The code (known as **kind**) and the string are made available to subscribers when they receive the notification. The notification is sent to all processes that are subscribed to the location at the time of the posting.

Notification

A notification specifies a database location and two additional user-defined items: a signed 32-bit integer and a null-terminated C string. These items are passed to the receiving process. The string is limited to 16,383 characters. If a notification sends a null string (**0**), it is received as an empty string (**""**).

When a notification is posted, a message is sent to the ObjectStore Server. The Server then matches the notification with subscriptions and queues messages to be sent to the receiving processes. The Server returns the number of messages queued. Notifications then proceed asynchronously to the Cache Manager of the receiving processes.

Range of Locations

A range of locations must be contiguous. You can specify a range to be an entire database, a segment, a cluster, an object or a range of objects, or a portion of an object. You can subscribe to a range of locations (that is, every location in a segment, or an array of objects), or to a single location. A notification, however, is always associated with a single database location.

Subscription

A client can subscribe to many ranges of database locations simultaneously. Subscriptions are stored in the ObjectStore Server for as long as the corresponding database is open by the client.

You can unsubscribe to ranges just as you can subscribe to them. The unsubscription is immediate. When you close a database, that unsubscribes all notifications for the database.

Notification Queuing

The Cache Manager maintains a queue of notifications for each client on a machine.

Nothing forces a client to read notifications. A client could choose to subscribe to notifications but never receive any.

To avoid resource exhaustion in the Cache Manager, the size of the notification queue for each client is fixed. If a notification is received when the client's queue is full, it is discarded. This is called an *overflow*.

Overflows do not cause any exception to be signaled and do not cause the application, the Cache Manager, or the Server to crash.

The Cache Manager keeps statistics on the notification queue that include

- Queue size
- Number of pending notifications
- Number of overflows

These statistics are made available to clients through the use of an API. The API, **os_dbutil::cmgr_stat()**, is described in Managing Cache Managers in Chapter 10 of the *ObjectStore Advanced C++ API User Guide*.

Receiving Notifications

Notifications are received in response to a call to **os_ notification::receive()**. This function can also be used to wait for notifications. Applications can poll for notifications without retrieving them using **os_notification::queue_status()**.

Notification Retrieval Alternatives

There are two main methods of retrieving notifications. One relies on a thread whose sole purpose is to receive notifications. The other method requires the application to poll to determine if notifications have arrived.

UNIX

A third method available on UNIX systems is called file-descriptor-based retrieval. This method works on both threads and nonthread systems.

Thread-Based Notification Retrieval

Using this method, a dedicated thread is started specifically to receive notifications. This thread calls **os_notification::receive()** with arguments specifying wait forever. When it receives a notification, it performs an application-specific action. For example, it might post a Windows message, modify the application’s transient data structures, or otherwise queue the notification for processing by another thread. It then waits for the next notification. The notification thread typically does very little work. It might do queue management, for example, maintaining a priority queue of notifications for another thread, or coalescing similar notifications. However, processing should be minimal, so the Cache Manager notification queue does not overflow.

In contrast to most other ObjectStore APIs, **os_notification::receive()** and **os_notification::queue_status()** are not locked out when other threads are in ObjectStore operations. If the thread does not access persistent data or call other ObjectStore APIs, it can run entirely asynchronously.

Polling-Based Notification Retrieval

Using this method, the application periodically polls to see if notifications have arrived. It does so using **os_notification::queue_status()** or **os_notification::receive()**. The application can do this polling in a main loop, or under control of timers or similar features provided by the environment. This mechanism is less flexible and less efficient than thread-based notification retrieval, but it is a reasonable option on platforms not offering threads. There are situations where polling can be quite efficient. If you are uncertain about the conditions affecting the level of efficiency, contact Object Design's Consulting Services group for assistance, or consult a programming text such as *UNIX Network Programming* by W. Richard Stevens. UNIX systems that do not support threads can make use of File-Descriptor-Based Notification Retrieval as described below.

The application should only check whether notifications have arrived. The application should not wait indefinitely (forever) for a notification, because it might be holding a lock, and the application expected to send the notification might be waiting for that lock. By waiting forever for a notification, you could create a deadly embrace. The Server's deadlock-detection mechanism cannot detect this.

File-Descriptor-Based Notification Retrieval

UNIX

On all UNIX platforms, ObjectStore can provide a file descriptor on which notifications arrive. This feature is not currently available on Windows or OS/2 platforms.

The clients poll or wait for notifications using the operating system functions **select()** or **poll()**. This is particularly useful in nonthreaded environments, where applications are designed to wait for multiple events by doing a multiplexed wait for activity on a set of file descriptors (**fd**s). Many Motif implementations, for example, fall into this category.

General Notification Behavior

The following sections describe the main characteristics of notification.

Subscribing and Unsubscribing

Subscribing is accomplished by means of static member functions of class **os_notification**. See the *ObjectStore C++ API Reference* description of the **os_notification** class.

You can unsubscribe to ranges just as you can subscribe to them. The unsubscription is immediate.

Discarded subscriptions

Closing a database for any reason unsubscribes all notifications for the database; that is, all the subscriptions are discarded. Therefore, it is the application’s responsibility to reinstitute the subscriptions.

Asynchronous processing

Notifications are processed asynchronously. After unsubscribing, notifications that might already be queued based on previous subscriptions might result in a client’s receiving notifications even after unsubscribing. ObjectStore makes no guarantees whether such notifications will be received or not.

Transactions

Transactions are entirely independent of immediate notifications, subscriptions, unsubscriptions, and notification retrieval. Sending of commit-time notifications is closely integrated with transactions. Commit-time notifications can only be queued inside a transaction, and they are only sent if

- No enclosing transaction aborts.
- The top-level enclosing transaction commits.

There are no restrictions on transaction types. The enclosing transactions might be read-only or update, local or global. Databases can be opened read-only, read / write, or for *multiversion concurrency control*, or *MVCC*, which is described in Chapter 2, Advanced Transactions, of *ObjectStore Advanced C++ API User Guide*.

Database changes made by an application are not visible to other applications until the enclosing top-level transaction commits.

Therefore, notifications that indicate changes to persistent data should generally be made at commit time.

In a two-phase commit transaction, either all notifications are queued for delivery (if the transaction commits), or none are (otherwise).

Security

In order to send and subscribe to notifications, you must open the database in question. Consequently, if a client does not have access to open a database, it cannot send or receive notifications associated with it.

Within a database, notifications are not integrated with ObjectStore security. A client can subscribe and notify based on database locations in any segment, even if it does not have access to the segment itself.

Performance Considerations

All notifications and subscriptions on a database go to the ObjectStore Server. The Server routes notifications to interested clients, and the Cache Manager queues the notifications for all its clients. Because the Server acknowledges each notification, sending a notification requires a round-trip message to the Server.

Polling for incoming messages only accesses shared memory and is very fast. If a client does not retrieve its notifications, the Cache Manager can run out of queue space.

Every call to **os_notification::subscribe**, **os_ notification::unsubscribe**, and **os_notification::notify_immediate** costs the client one round-trip message to the ObjectStore Server. Polling for notifications using **os_notification::receive** results in a call to **poll()** or **select()** or other operating-system-specific call. Polling for notifications using **os_notification::queue_status** is a simple access to shared memory and is the fastest polling mechanism.

If any commit-time notifications are queued during a transaction, there is an additional RPC call to the ObjectStore Server during the top-level commit operation.

Notifications are stored and forwarded in the Server, Cache Manager, and sometimes even in the receiving application. Therefore, delivery of notifications might not be particularly fast. Performance varies according to system load and the amount of notification processing. For example, delivery could range from milliseconds to several seconds.

As a general rule, if you plan your application to use notification, you should not expect high throughput. Do not expect a client application to send or receive more than about 10 notifications per second.

Event validation

ObjectStore does not check events for validity. It is possible to specify an address in a notification that is illegal in another process. For example, you could allocate a new object and post an immediate notification using its location. Other processes see this address as invalid because the new object has not yet been committed.

Restriction on use with access hooks

Notification APIs cannot be called from within *access hooks*. Information about access hooks can be found in the discussion on **os_database::set_access_hooks()** in the *ObjectStore C++ API Reference*.

Notification Usage

The guidelines for sending and receiving notifications are summarized in the next paragraphs.

Sending notifications

The main class is **os_notification**. You must include the **ostore.hh** file, and link with **-los** on UNIX, and **OSTORE.LIB** on Windows and OS/2 systems. The signature of the function that sends a notification is

```
/* $OS_ROOTDIR/include/ostore/client/client.hh */

os_notification::notify_immediate
(os_reference&, int kind=0, const char* message=0);
```

To ensure that subscribers do not receive notifications until the changes are visible in the database, use **os_notification::notify_on_commit**.

Receiving notifications

An application receives notification using the following functions:

```
os_notification::receive (os_notification*&, int timeout=-1);

os_reference os_notification::get_reference();
```

int os_notification::get_kind();

const char* os_notification::get_string();

Be sure to delete the returned heap-allocated **os_notification** object when done.

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 Chapter 1, Overview of Managing ObjectStore, in *ObjectStore Management* for specific information about defaults.

Notification Errors

The notification APIs do not do complete validation of the arguments passed to them. Invalid arguments can therefore cause segmentation violations or other undefined behavior. See the *ObjectStore C++ API Reference*, Appendix A, Exception Facility, for information on specific errors.

ObjectStore Utilities for Managing Notification

The **ossvrstat** utility displays statistics on the number of notifications received and sent by the Server.

The **oscmstat** utility displays information on notifications queued for clients. This is useful in debugging applications that use notifications.

Detailed descriptions of these and other ObjectStore utilities can be found in Chapter 4, Utilities, in *ObjectStore Management*.

Notifications Example

The following example illustrates the use of notifications.

```
#include <ostore/ostore.hh>
#include <iostream.h>
#include <assert.h>

int main(int argc, char** argv) {

    const char* db_name = "notif.db";
    const char* root_name = "Test Object";

    /* can see client name with "ossvrstat -clients <host>" */
    objectstore::set_client_name(argv[0]);
    objectstore::initialize();

    cout << "Opening database "<< db_name << endl;
    os_database* db = os_database::open(db_name,0,0644);

    os_reference ref1 = 0;
    os_reference ref2 = 0;

    os_transaction* txn =
        os_transaction::begin(os_transaction::update);
    os_database_root* root = db->find_root(root_name);
    if (!root) {
        cout << "Creating a couple of ints" << endl;
        root = db->create_root(root_name);
        root->set_value(new (db, os_typespec::get_int(), 2) int[2]);
    } /* end if */

    ref1 = root->get_value();    /* &int[0] */
    ref2 = &((int*)ref1.resolve())[1];     /* &int[1] */
    txn->commit();
    delete txn;

    os_notification* note;
    int iterations = 0;

    os_transaction::begin(os_transaction::read_only);

    /* the Initiator process takes no args on command line */
    if ( argc == 1) {
        cout << "Initiator Starting notifications..." << endl;

        /* subscribe to ref2; */
        os_notification::subscribe(ref2);

        while (iterations < 10) {
            iterations++;
            cout << "sending notification, kind = "<< iterations << endl;
            /* send immediate notification on ref1 with iterations */
            /* as kind */
            os_notification::notify_immediate(ref1,iterations);
            /* now get response into note */
            os_notification::receive(note);

            /* make sure note response is on ref2 */
            os_reference ref = note->get_reference();
            assert(ref == ref2);

            /* make sure correct iterations comes back */
            int kind = note->get_kind();
            assert(kind == iterations);

            delete note;    /* avoid memory leak */

            sleep(2);
        } /* end while */

        /* Tell Responder to exit by sending
        notification on ref1 with kind=0 */
        cout << "sending notification, kind = 0" << endl;
        os_notification::notify_immediate(ref1,0);

        /* Initiator done */

    } /* end if */
    else {

        /*  the Responder process takes any args on command line */
        cout << "Responder Waiting for Notifications" << endl;

        /* subscribe to ref1 */
        os_notification::subscribe(ref1);

        while(1) {
            /* receive notification for ref1 into note */
            os_notification::receive(note);

            /* see what kind it is */
            int kind = note->get_kind();

            cout << "received notification, kind = "<< kind << endl;
            /* if kind is 0, exit */
            if (kind == 0)
            break;    /* Responder done */

            /* make sure notification is about ref1 */
            os_reference ref = note->get_reference();
            assert(ref == ref1);

            delete note;   /* avoid memory leak */

            /* send notification on ref2 with kind */
            os_notification::notify_immediate(ref2,kind);

        } /* end while */

    } /* end if */

    return 0;
}
```

Chapter 5
Collections

The information about collections is organized in the following manner:

Collections Overview

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

In order to implement collection functionality, the ObjectStore collection facility provides

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

Collection Class Library

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.

Because collections are pointers to objects — rather than the objects themselves — an object can be contained in many different collections. Furthermore, collections can be used in transient or persistent memory, depending on the needs of your application.

Collection Query and Manipulation Features

Collections form the basis of the ObjectStore query facility, which allows you to select those elements of a collection that satisfy a specified condition. Queries with simple conditions are discussed in this chapter. Queries with complex conditions are described in Chapter 5, Queries and Indexes, of the *ObjectStore Advanced C++ API User Guide*.

The ObjectStore collection facility gives you a great deal of control over the behavior and representation of the collections you create. Other collection facilities allow you to iterate over the elements in a collection, and to query collections for elements meeting simple or sophisticated sorting criteria.

This allows you, for example, to create either ordered or unordered collections, and collections that either do or do not allow duplicates. You can also choose from among a group of system-supplied collection representations, such as hash tables and packed lists. You can even specify how a collection's representation is to change in response to changes in the collection's size.

Collections are commonly used to model many-valued attributes, and they can also be used as class extents (which hold all instances of a particular class). Collections of one type — *dictionaries* — associate a key with each element or group of elements, and so can be used to model binary associations or mappings. ObjectStore dictionaries are described in detail in Dictionaries on page 139.

Requirements for Applications Using Collections

Note the following requirements for using ObjectStore collections:

- All applications that use collections must include the collections header files and initialize the collections facility.
- They must also generate schema using the collections / queries library schemas and link in the collections / queries libraries.
- Applications using dictionaries must also mark each type of dictionary in a schema source file.

Include Files

Programs that use ObjectStore collections must include the header file **<ostore/coll.hh>** after including the standard ObjectStore header file **<ostore/ostore.hh>**.

If your application uses ObjectStore dictionaries, your program must include **<ostore/coll/dict_pt.hh>** and must also include **<ostore/coll/dict_pt.cc>** in any source file that instantiates an **os_Dictionary**, following the other header files. See ObjectStore Header Files in Chapter 2 of *ObjectStore Building C++ Interface Applications*.

Initializing the Collection Facility

Any program using collection functionality must first call the static member function **os_collection::initialize()**. Call this function after calling **objectstore::initialize()**. For example:

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

Linking

Programs that use ObjectStore collections must also link with the appropriate ObjectStore collections libraries and library schema. Collections library names are platform specific:

	UNIX Platforms	*Windows and OS/2 Platforms*
Collections	**-loscol liboscol.so**	**ostore.lib**
	liboscol.ldb	**os_coll.ldb**

	UNIX Platforms	*Windows and OS/2 Platforms*
Indexes and queries	**-losqury libosqury.so** **libqry.ldb**	**os_query.ldb**

Using Persistent Collections

If you use persistent collections (whether parameterized or not) the actual representation that is used for storage in the database is an **os_collection** internal representation. You do not have to mark any collection type in your schema source file.

Using Persistent Dictionaries

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

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

Specific information about marking dictionaries can be found in Marking Persistent Dictionaries on page 139. The **OS_MARK_DICTIONARY()** macro is described in the *ObjectStore Collections C++ API Reference*.

Thread Locking

If your application does not use multiple threads, disable collections thread locking by passing **0** to the following member of the class **os_collection**:

static void set_thread_locking(os_boolean) ;

If your application uses multiple threads, see Chapter 3, Threads, in the *ObjectStore Advanced C++ API User Guide*.

Introductory Collections Example

Here is a simple example to illustrate how and when to use collections. ObjectStore collections provide some alternatives to linked lists, C++ arrays, and other aggregation data structures. In cases where your application uses functionality such as queries and ranges, collections are easier to use and more powerful.

Note, though, that because the functionality is so rich and varied, the collections facilities add overhead to your code and database; if your application does not require collections, a simple linked list you can write yourself might be a more suitable choice.

Using the Example: Linked List of Notes on page 40 as a base, the following example illustrates how to use collections.

Header file: **note.hh**

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

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

class note {

    public:

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

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

Main program: **main.cc**

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

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

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

main(int argc, char** argv) {

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

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

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

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

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

        if(!notes) {
            notes = &os_list::create(db);
            root_head->set_value(notes);
        } /* end if */

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

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

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

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

    db->close();
}
```

Choosing a Collection Type

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

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

For information on

- Differences between templated (parameterized) and nontemplated (nonparameterized) collection classes, see Templated and Nontemplated Collections on page 116.
- Characteristics of ObjectStore collection classes — such as their representations, sizes, and default behaviors — see Collection Characteristics and Behaviors on page 112.
- A hierarchical representation of the relationships between the ObjectStore collection types, see the Class Hierarchy Diagram on page 112.
- How to create collection classes, see Creating Collections on page 119.

os_Set and os_set

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

Besides lacking order, something that distinguishes sets from some other types of collections is that they do not allow *multiple occurrences* of the same element. This means that inserting a value that is already an element of a set either leaves the set unchanged or causes the signaling of a run-time exception (depending on the behavior you have specified for the set). In either case, sets disallow duplicates.

os_Bag and os_bag

Bags are collections that not only keep track of what their elements are, but also of the number of occurrences of each element. In other words, bags allow duplicate elements. The class **os_Bag** provides all the operations available for sets, as well as an operation, **count()**, that returns the number of occurrences of a given element in a given collection.

os_List and os_list

In addition to sets and bags, the ObjectStore collection facility supports *lists*, collections that associate a numerical position with each element based on insertion order. Lists can either allow or disallow duplicates (by default they allow duplicates). In addition to simple insert (insert into the beginning or end of the collection) and simple remove (removal of the first occurrence of a specified element) you can insert, remove, and retrieve elements based on a specified numerical position, or based on a specified *cursor* position (see Accessing Collection Elements with a Cursor or Numerical Index on page 132).

os_Collection and os_collection

Of the collection types, **os_Collection** (and **os_collection**) offer the most flexibility in making behavior changes during the lifetime of an instance. Creating an instance of the base class **os_Collection** gives you direct control over allowing duplicate elements and maintaining element order, the behaviors that distinguish sets, bags, and lists.

The **os_Collection** class permits you to change these and other behaviors mentioned above for an **os_Collection**. This means that an instance of **os_Collection** could, at one point in its lifetime, have set-like behavior, and at another point have bag-like or list-like behavior.

os_Array and os_array

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

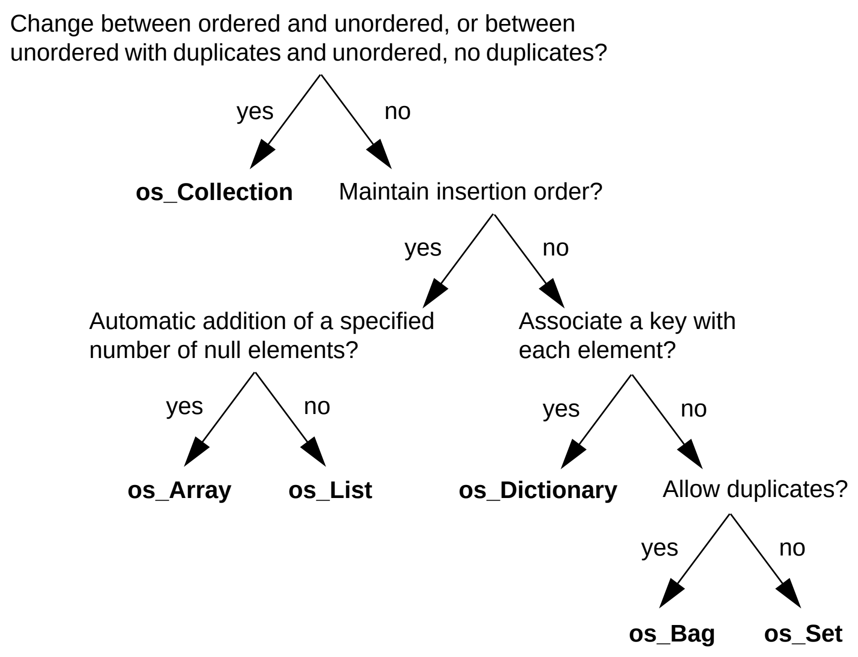
os_Dictionary and os_rDictionary

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

Dictionaries are somewhat different from other ObjectStore collection classes in their use of keys. See Dictionaries on page 139 for additional information on how dictionaries differ from other kinds of ObjectStore collections.

Using a Decision Tree to Select a Collection Type

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


Change between ordered and unordered, or between unordered with duplicates and unordered, no duplicates?
yes
no
os_Collection
Maintain insertion order?
yes
no
Automatic addition of a specified number of null elements?
Associate a key with each element?
yes
no
os_Array
os_List
yes
no
os_Dictionary
Allow duplicates?
yes
no
os_Bag
os_Set

Collection Characteristics and Behaviors

Collections Store Pointers to Objects

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

Collections Can Be Transient or Persistent

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

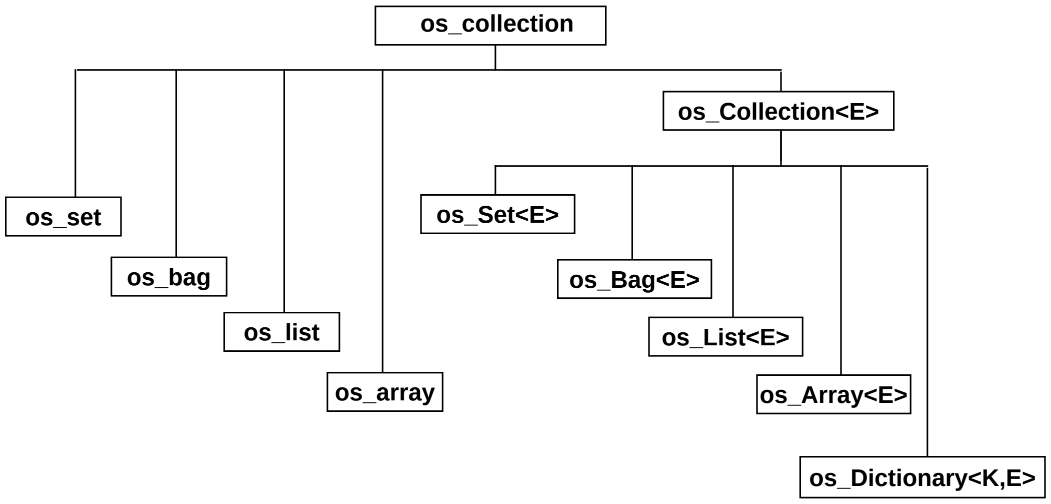
Parameterized and Nonparameterized Collections

Every ObjectStore collection class (except **os_Dictionary**) is provided in both a templated (parameterized) and a nontemplated (nonparameterized) form. See Templated and Nontemplated Collections on page 116.

Templated classes use uppercase letters in their class names (**os_ Set**), whereas nontemplated classes use lowercase letters in their class names (**os_set**).

Class Hierarchy Diagram

The following diagram shows the hierarchical relationship among all the ObjectStore collection classes. Note that **E** is actually a pointer value: the *element type parameter* used by the templated collection classes to specify the types of values allowable as elements. See Using Collections with the Element Type Parameter on page 116 for more information.



Collection Behaviors

The ObjectStore collection classes vary according to what behaviors and characteristics are permitted, prohibited, or permitted under some circumstances. The table below identifies default settings and behavior; however, you can customize many of these settings. You can, for example, change the default size of a collection and, in some cases, you can specify whether or not null elements can be inserted into the collection. See Chapter 4, Advanced Collections, of the *ObjectStore Advanced C++ API User Guide* for more information on customizing your ObjectStore collections.

Collections Class	***Maintain Element Order***	***Allow Duplicates***	***Signal Duplicates***	***Allow Nulls***
os_Set	Forbidden	Forbidden	Off by default	Off by default
os_Bag	Forbidden	Required	Forbidden	Off by default
os_List	Required	On by default	Off by default	Off by default
os_Collection	Off by default	Off by default	Off by default	Off by default
os_Array	Required	On by default	Off by default	Required
os_Dictionary	Forbidden	Required	Forbidden	Required

Note that **os_Dictionary** differs substantially from other collections classes in its behaviors. Dictionary behaviors are related to the *key* of an element rather than to an element itself. See Dictionaries on page 139 for information on how ObjectStore dictionaries differ from other ObjectStore collection classes. See also Creating Dictionaries on page 121.

Expected Collection Size

By default, all collection classes are presized with a representation suitable for a size of less than 20. The **expected_size** argument for the collection **create()** functions lets you supply a different default size. For more information, see Specifying Expected Size in Chapter 4 of the *ObjectStore Advanced C++ API User Guide*.

Performing pick() on an Empty Set

For all collection classes, performing **pick()** on an empty collection or on an empty result of a query of a list or collection raises an err_coll_empty error.

Collection Representations

The default representation policies for ObjectStore collections differ depending on whether the collection is created with the **create()** function or is embedded within another object.

Collections created by **create()**

The representation types listed in the following table are described in detail in Chapter 4, Advanced Collections, of the *ObjectStore Advanced C++ API User Guide*.

Collections Class	*Size 0 to 20*	*Greater Than 20*
os_Set	os_chained_list	os_dyn_hash
os_Bag	os_chained_list	os_dyn_bag
os_List	os_chained_list	os_packed_list
os_Collection	os_chained_list	os_dyn_hash
os_Array	os_chained_list	os_packed_list

Representations for embedded collections

The representation types listed in the following table are described in detail in Chapter 4, Advanced Collections, of the *ObjectStore Advanced C++ API User Guide*.

Collections Class	***Size 0 to 20***	***Greater Than 20 (Do Not Maintain Cursors)***	***Greater Than 20 (Maintain Cursors)***
os_Set	**os_chained_list**	**os_dyn_hash**	Not applicable
os_Bag	**os_chained_list**	**os_dyn_bag**	Not applicable
os_List	**os_chained_list**	**os_packed_list**	**os_packed_list**
os_Collection	**os_chained_list**	**os_dyn_hash**	**os_packed_list**
os_Array	**os_chained_list**	**os_packed_list**	**os_packed_list**

Note that **expected_size** determines the collection's initial representation. So, for example, if you set the **expected_size** for an **os_Set** to 21, **os_dyn_hash** is used for an **os_Set**'s collection's entire lifetime. (This can, however, be customized; see Customizing Collection Representation in Chapter 4 of the *ObjectStore Advanced C++ API User Guide*).

Representations used for **os_Dictionary** are discussed separately in Dictionary Representation on page 141.

Templated and Nontemplated Collections

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

Using Collections with the Element Type Parameter

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

```
os_Set<part*> &a_part_set = os_Set<part*>::create(db1) ;
```

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

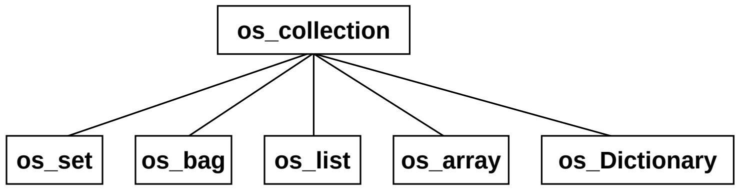
Example: **os_Set**

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

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

class part {

    public:

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

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

};
```

Using Collections Without Parameterization

You can choose to use the following nonparameterized collection classes.



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

Difference between parameterized and nonparameterized interfaces

The difference between the parameterized and nonparameterized interfaces is that with the parameterized interface you can inform the compiler of the type of element a collection is to have, allowing the compiler to provide an extra measure of type safety. With the nonparameterized interface, elements are always typed as **void*** pointers.

Most of the examples in this manual use the parameterized interface, but if you are not using parameterization, just drop any construct of the form

<*element-type-name***>**

and use the nonparameterized collection type names (beginning with **os_** followed by a lowercase letter). The **os_Set** class definition example in Using Collections with the Element Type Parameter on page 116 would look like this:

Example: **os_set()**

```
class employee ;
extern os_database *db1 ;

class part {

    public:

        int part_number ;
        os_set &children ;
        employee *responsible_engineer ;

        part (int n) :
            children(os_set::create(db1)){
                part_number = n;
                responsible_engineer = 0 ;
            }
    };
```

The **<part*>** is left out after each occurrence of **os_Set**, and **os_Set** is changed to **os_set**.

Nonparameterized collections are typed **void***

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

Creating Collections

This section presents information on creating collections. The **os_Array**, **os_Bag**, **os_Collection**, **os_List**, and **os_Set** classes are discussed together in the first subsection. **os_Dictionary** is discussed separately in the following subsection.

General Guidelines

You can create collections for each collections class with the following functions:

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

The create method is a wrapper for constructors. For example:

```
static os_Collection<E> & os_Collection<E*>::create(
        os_database *db,
        os_unsigned_int32 behavior = 0,
        os_int32 expected_size = 0
)
```

These wrappers return a reference to a new, empty collection.

Do not use **new** to create collections.

Generally, it is preferable to use the **create()** function for each type of collection, as it results in better performance and better locality of reference because of its underlying optimization for mutation.

Each collection class has a **destroy()** function that deletes a specified collection. See Destroying Collections on page 123 for more information. You can also call **delete()** to destroy a collection created with **::create()**.

Use a constructor only for stack-based collections or collections embedded directly within another class. However, avoid embedding collections directly. Instead, embed collections as pointers or references, and call the **create()** function in a constructor initialization list.

create() functions

Each collection function has four overloadings, as described in the *ObjectStore Collections C++ API Reference*. The first argument, which specifies where to allocate the new collection, is the only required argument. Depending on the overloading, **create()** functions for collection types specify a database, segment, object cluster, or proximity to another object.

All other arguments are optional, because they are declared with default values.

Constructor functions

The overloadings of constructor functions are listed in the *ObjectStore Collections C++ API Reference*. For each constructor, the first two overloadings create an empty set. The second overloading lets you specify the expected collection size (see Expected Collection Size on page 114). The last two overloadings are copy constructors, creating a collection with the same membership as another specified collection.

Customizing collections

Many of the characteristics and behaviors of various types of collections can be modified. For general information about each class, see the following sections in the *ObjectStore Collections C++ API Reference*.

Collection Type	*Parameterized Class*	*Nonparameterized Class*
Array	**os_Array**	**os_array**
Bag	**os_Bag**	**os_bag**
Collection	**os_Collection**	**os_collection**
List	**os_List**	**os_list**
Set	**os_Set**	**os_set**

To find out how to

- Modify the size of the set, see Specifying Expected Size in Chapter 4 of the *ObjectStore Advanced C++ API User Guide*.
- Modify the behavior of the set, see Customizing Collection Behavior in Chapter 4 of the *ObjectStore Advanced C++ API User Guide*.
- Modify a set’s representation, see Customizing Collection Representation in Chapter 4 of the *ObjectStore Advanced C++ API User Guide*.

Creating Dictionaries

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

You can create dictionaries with **os_Dictionary::create()** or an **os_ Dictionary** constructor. Use the constructor only for stack-based arrays, or arrays embedded directly within another object. If you want to use a reference-based dictionary, use the **os_rDictionary** functions.

os_Dictionary:: create()

There are four overloadings of **os_Dictionary::create()**, as described in the *ObjectStore Collections C++ API Reference*. The first argument, which specifies where to allocate the new collection, is the only argument required for **os_Dictionary::create()**. Depending on the overloading, it specifies a database, segment, or object cluster. All other arguments are optional, since they are declared with default values.

The **os_Dictionary::create()** overloadings require not only the *element type parameter* but the *key type parameter* as well. See Using Collections with the Element Type Parameter on page 116 for information about the element type parameter, and “Key type parameter”, below, for information about the key type parameter.

os_Dictionary:: os_Dictionary()

The **os_Dictionary::os_Dictionary()** constructor is described in the *ObjectStore Collections C++ API Reference*. Use the dictionary constructor only to create stack-based dictionaries, or dictionaries embedded within other objects. As an alternative to embedded dictionaries, consider using a reference or pointer, which allow you to use **os_Dictionary::create**.

In general, it is preferable to use **os_Dictionary::create()**, as it results in higher performance and better locality of reference because of its underlying optimization for mutation.

Key type parameter

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

Integer keys

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

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

Class keys

For class keys, the class must have a destructor that zeroes any pointers it contains, a no-arg constructor, and **operator=**.

void* keys

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

char* keys

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

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

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

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

Destroying Collections

Each collection class has a static member for deleting a specified collection.

static void os_Collection::destroy(os_Collection<E>&) ;

static void os_Set::destroy(os_Set<E>&) ;

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

static void os_List::destroy(os_List<E>&) ;

static void os_Array::destroy(os_Array<E>&) ;

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

Destroying a collection class does *not* destroy the elements in the class; it deletes the specified collection itself and deallocates its associated storage. You can either call **destroy()** or simply delete the collection with the delete pointer.

All the collection class **destroy()** functions are described in Chapter 2, Collection, Query, and Index Classes, of the *ObjectStore Collections C++ API Reference*.

Inserting Collection Elements

You update collections by inserting and removing elements, or by using the assignment operators (see Copying, Combining, and Comparing Collections on page 137). The insert operations for **os_Collection** and its subtypes are declared this way:

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

For more information on behavioral enumerators

Refer to Customizing Collection Behavior in Chapter 4 of the *ObjectStore Advanced C++ API User Guide* for descriptions of the behavior enumerators (such as **signal_dup_keys**, **allow_duplicates**, and **signal_duplicates**) that are mentioned in this section.

Inserting Dictionary Elements

For dictionaries, you specify an *entry*, that is, a key, along with the element to be inserted. So **os_Dictionary::insert()** is declared this way:

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

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

Caution

For dictionaries with **signal_dup_keys** behavior, if an attempt is made to insert something with the same key as an element already present, err_am_dup_key is signaled.

Duplicate Insertions

For collections without **allow_duplicates** and **signal_duplicates** behavior, inserting something that is already an element of the collection leaves the collection unchanged.

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

For collections with **signal_duplicates** behavior, inserting a duplicate raises err_coll_duplicate_insertion.

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

Null Insertions

If you insert a null pointer (**0**) into a collection without **allow_nulls** behavior, the exception err_coll_nulls is signaled.

Ordered Collections

Inserting into a collection with **maintain_order** behavior adds the element to the end of the collection. See also Accessing Collection Elements with a Cursor or Numerical Index on page 132.

Duplicate Keys

For dictionaries with **signal_dup_keys** behavior, if an attempt is made to insert something with the same key as an element already present, err_index_duplicate_key is signaled.

Changing a Collection's Behavior

You can change many of a collection's behavioral characteristics. See Customizing Collection Behavior in Chapter 4 of the *ObjectStore Advanced C++ API User Guide*.

Changing a Collection's Representation Policy

You can change a collection's representation policy at any time, but keep in mind that changing a representation essentially reallocates a new object and copies the elements from the old representation to the new representation. See Customizing Collection Representation in Chapter 4 of the *ObjectStore Advanced C++ API User Guide*.

Removing Collection Elements with **remove()**

The remove operations for **os_Collection** and its subtypes are declared this way:

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

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

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

Ordered Collections

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

Removing Dictionary Elements

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

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

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

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

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

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

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

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

Caution regarding duplicate insertions

For dictionaries with **signal_dup_keys** behavior, if an attempt is made to insert something with the same key as an element already present, err_am_dup_key is signaled. Refer to Customizing Collection Behavior in Chapter 4 of the *ObjectStore Advanced C++ API User Guide* for information about **signal_dup_keys.**

Testing Collection Membership with **contains()**

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

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

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

Dictionaries

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

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

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

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

Finding the Count of an Element with **count()**

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

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

If **e** is not an element of the collection, **0** is returned.

Dictionaries

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

```
os_unsigned_int32 count_values(const K const &key_ref) const ;
os_boolean contains(const K *key_ptr) const ;
```

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

Finding the Size of a Collection with **cardinality()**

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

os_unsigned_int32 cardinality() const ;

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

Checking for an Empty Collection with empty()

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

os_int32 empty() ;

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

Using Cursors for Navigation

The ObjectStore collection facility provides a number of classes that help you navigate within a collection. The **os_Cursor** class, the **os_index_path** class, and the **os_coll_range** class all help you insert and remove elements, as well as retrieve particular elements or sequences of elements.

- The **os_index_path** class is described in Using Paths in Navigation in Chapter 4 of the *ObjectStore Advanced C++ API User Guide*. See also **os_index_path** in the *ObjectStore Collections C++ API Reference*.
- The **os_coll_range** class is described in Using Ranges in Navigation in Chapter 4 of the *ObjectStore Advanced C++ API User Guide*. See also **os_coll_range** in the *ObjectStore Collections C++ API Reference*.

The **os_Cursor** class is discussed here and in the immediately following sections of this chapter. See also **os_Cursor** in the *ObjectStore Collections C++ API Reference*.

Cursors

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

When you create a vanilla cursor with no index path or collection range, you specify its associated collection, and the cursor is positioned at the collection's first element. With members of **os_Cursor**, you can reposition the cursor, as well as retrieve the element at which the cursor is currently positioned. See Traversing Collections with Default Cursors on page 134.

Some members of the collection classes take cursor arguments. These functions support insertion, removal, and replacement of elements. See Accessing Collection Elements with a Cursor or Numerical Index on page 132.

Accessing Collection Elements with a Cursor or Numerical Index

Getting positional access within a collection

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

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

The cursor-based overloadings must use a default vanilla cursor. The cursor-based overloadings of **remove_at()**, **replace_at()**, and **retrieve()** can also be used for unordered collections. (See Traversing Collections with Default Cursors on page 134 for additional information.)

Manipulating first and last elements in a collection

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

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

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

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

Traversing Collections with Default Cursors

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

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

Example: iterating through a collection with **os_Cursor**

```
os_database *db1 ;
. . .
os_Collection<person*> &people
    = os_Collection<person*>::create(db1) ;

. . . /* insertions into people */

os_Collection<person*> &teenagers
    = os_Collection<person*>::create(db1);

person* p;

os_Cursor<person*> c(people);

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

The **for** loop in this example retrieves each element of the collection **people** and adds those between the ages of 13 and 19 to the collection **teenagers**.

Creating Default Cursors

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

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

Its constructor takes a **Collection&** (**people** in the example above) as argument. This is the collection to be traversed. The traversal proceeds in an arbitrary order for unordered collections and, for ordered collections, in the order in which the elements were inserted. See also Controlling Traversal Order, Performing Collection Updates During Traversal, and Restricting the Elements Visited in a Traversal in Chapter 4 of the *ObjectStore Advanced C++ API User Guide*

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

os_Cursor's parameter (**person*** in the example) indicates the type of elements in the collection being traversed. So the cursor's parameter must be the element type (see Using Collections with the Element Type Parameter on page 116) of the collection passed as the constructor argument.

os_Cursor::first()

The program then has a **for** loop. The traversal is performed with a **for** loop. The initialization part of the loop header is an assignment involving a call to the member function **os_Cursor::first()**:

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

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

os_Cursor::next()

The incrementation part of the **for** loop header is an assignment involving a call to the member function **os_Cursor::next()**:

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

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

os_Cursor::more()

The loop's condition is a call to the member function **os_Cursor::more()**:

```
c.more()
```

This function returns a nonzero 32-bit integer (true) when the cursor is still positioned at some element of the collection, and **0** (false) when it is null.

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

Alternative to using **more()**

For collections that do not allow null elements, you can take advantage of the fact that **first()** and **next()** return null pointers when there is no first or next element. This means you can use the values returned by these functions (in this case **p**) as the loop condition (as long as the collection contains no null pointers — see Customizing Collection Behavior in Chapter 4 of the *ObjectStore Advanced C++ API User Guide*).

```
os_database *db1 ;
. . .
os_Collection<person*> &people
    = os_Collection<person*>::create(db1) ;
. . .    /* inserts to people */
os_Collection<person*> &teenagers
    = os_Collection<person*>::create(db1) ;
person* p ;
os_Cursor<person*> c(people) ;
for ( p = c.first() ; p ; p = c.next() )
    if ( p–>age >=13 && p–>age <= 19 )
        teenagers.insert(p) ;
```

Rebinding Cursors to Another Collection

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

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

Once rebound, the cursor is positioned at the specified collection's first element.

This last overloading is for rebinding cursors whose order is specified by a rank function. See Path-Based Traversal in Chapter 4 of the *ObjectStore Advanced C++ API User Guide*.

Copying, Combining, and Comparing Collections

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

Collection comparison operators

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

Dual Purpose of the Operators

All these operators have a dual purpose. They can be used on two collections, or they can be used on a collection (as left-hand operand) and an instance of that collection’s element type (as right-hand operand). For example, they can be used on a set of parts and a part. In that case, the instance of the collection’s element type is treated as a collection whose one and only element is that instance. Performance varies by representation.

So, for example, you can use the union equals operator, **|=**, as a convenient way of performing inserts:

```
os_Collection<message*> &a_set
    = os_Collection<message*>::create(db1) ;
message *a_msg ;
. . .
```

```
a_set |= a_msg ;
```

And you can use **-=** to remove elements:

```
a_set -= a_msg ;
```

Ordered Collections and Collections with Duplicates

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

```
big_list |= little_list ;
```

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

```
the_copy = the_original;
```

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

In general, the update operators (**=**, **|=**, **-=**, **&=**) bundle together a sequence of inserts or removes of elements of one or more operands *in the order in which those elements appear in the operands*, the default iteration order for the operands. This describes only the *behavior* of the operators. The implementations might be different.

For example, to add all of a part's children to a given set, you might do

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

This is behaviorally equivalent to

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

Dictionaries

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

Required include files

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

Marking Persistent Dictionaries

The **OS_MARK_ DICTIONARY()** macro

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

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

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

Example: schema file

```
/* schema.cc */

#include <ostore/ostore.hh>
#include <ostore/coll.hh>
#include <ostore/coll/dict_pt.hh>
#include <ostore/manschem.hh>
#include "dnary.hh"

OS_MARK_DICTIONARY(void*,Course*) ;
OS_MARK_DICTIONARY(int,Employee**) ;
OS_MARK_DICTIONARY(int,Course*) ;
OS_MARK_SCHEMA_TYPE(Course) ;
OS_MARK_SCHEMA_TYPE(Employee) ;
OS_MARK_SCHEMA_TYPE(Department) ;
```

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

The **OS_MARK_DICTIONARY()** macro is described in *ObjectStore Collections C++ API Reference*.

Marking Transient Dictionaries

The **OS_TRANSIENT_ DICTIONARY()** macro

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

However, using **OS_TRANSIENT_DICTIONARY()** more than once with the same key type will result in a complication error. For example, the following will not compile correctly:

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

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

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

For related information on these macros, se the *ObjectStore Collections C++ API Reference*.

Dictionary Behavior

Every dictionary has the following properties:

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

By default a new dictionary also has the following properties:

- Performing **pick()** on an empty dictionary raises err_coll_empty.
- Duplicate keys are allowed; that is, two or more elements can have the same key.
- Range lookups are not supported; that is, key order is not maintained.

You can customize the behavior of new dictionaries with regard to these last three properties. For large dictionaries that maintain key order, there is also an option for reducing contention. See Customizing Collection Behavior in Chapter 4 of the *ObjectStore Advanced C++ API User Guide*.

Dictionary Representation

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

In addition to the key type and element type parameters, the class **os_rDictionary** also has a *reference type parameter*, whose actuals are ObjectStore reference types. Specify either **os_reference** or **os_ reference_version**.

The **os_rDictionary** class is described in the *ObjectStore Collections C++ API Reference*.

Visiting the Elements with Specified Keys

For dictionaries, you can specify a restriction that is satisfied by elements whose key satisfies a specified range.

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

An element satisfies this cursor’s restriction if its key satisfies **range**. If the dictionary’s key type is a class, you must supply rank and hash functions for the class. To do this, the dictionary must be created with the behavior **maintain_key_order**. See Customizing Collection Behavior in Chapter 4 of the *ObjectStore Advanced C++ API User Guide*.

Picking the Element with a Specified Key

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

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

These two differ only in that with one you supply a reference to the key, and with the other you supply a pointer to the key. Again, if there is more than one element with the key, an arbitrary one is picked and returned. If there is no such element and the dictionary has **pick_from_empty_returns_null** behavior, **0** is returned. If there is no such element and the dictionary does not have **pick_from_empty_returns_null** behavior, err_coll_empty is signaled.

Retrieving ranges of elements

For dictionaries, you can also retrieve an element whose key satisfies a specified collection range (see Specifying Collection Ranges in Chapter 4 of the *ObjectStore Advanced C++ API User Guide*) with

```
E pick(const os_coll_range&) const ;
```

For example:

```
a_dictionary.pick( os_coll_range(GE, 100) )
```

returns an element of **a_dictionary** whose key is greater than or equal to 100. The dictionary must have the behavior **os_ dictionary::maintain_key_order** for a **pick()** using an **os_coll_range**.

As with the other **pick()** overloadings, if there is more than one such element, an arbitrary one is picked and returned. If there is no such element and the collection has **pick_from_empty_returns_ null** behavior, **0** is returned. If there is no such element and the dictionary does not have **pick_from_empty_returns_null** behavior, err_coll_empty is signaled.

Retrieving elements when the key type is a class

If the dictionary’s key type is a class, you must supply rank and hash functions for the class. See Supplying Rank and Hash Functions in Chapter 4 of the *ObjectStore Advanced C++ API User Guide*.

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

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

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

Writing Destructors for Dictionaries

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

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

Here is an example where type K is class **myString**:

```
class myString
{
    private:
        char* theString;
        int  len;
}
RMString::RMString(char* theChars)
{
    if (theChars == 0)
        len = 0;
    else
        len = strlen(theChars);
    theString = new(os_segment::of(this),
    os_typespec::get_   char(),len + 1)
            char[len+1];
    if (theChars == 0)
        theString[0] = '\0';
    else
        strcpy(theString, theChars);
}

RMString::~RMString()
{
    delete[] theString;
/******************************************************
    The following line solves the multiple delete problem
*******************************************************/
    theString = 0;

}
```

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

```
Invalid argument to operator delete

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

Example: Using Dictionaries

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

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

Note that the **dnary.cc** example includes **<ostore/coll/dict_pt.cc>** *instead of* **dict_pt.hh**, and is needed since it contains the bodies of the functions declared in **dict_pt.hh**. You do not need to also include **dict_pt.hh** because it is included in **dict_pt.cc.** Finally, there is the file **schema.cc**, a schema source file.

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

Header file: **dnary.hh**

```
/* dnary.hh */

#include <ostore/ostore.hh>
#include <ostore/coll.hh>
#include <ostore/coll/dict_pt.hh>
#include <iostream.h>
#include <stdlib.h>

class Student ;
class Grade ;
class Course ;
class Student {

   public:
      int get_id() const ;
      const os_Collection<Course*> &get_courses() const ;
      int add_course( Course*, Grade* = 0 ) ;
      void remove_course(Course*) ;
      Grade *get_grade_for_course(const Course*) const ;
      void set_grade_for_course(Course*, Grade*) ;
      os_Collection<Course*> &get_courses_with_grade(
            const Grade* ) const ;
      float get_gpa() const ;
      static os_typespec *get_os_typespec() ;
```

```
        Student(int id, os_segment*);
        ~Student() ;

    private:
        int id ;
        os_Collection<Course*> & courses ;
        os_Dictionary<const void*, Grade* & course_grade ;
        os_Dictionary<void*, Course* & grade_course ;

} ;

class Grade {

    public:
        const char *get_name() const ;
        float get_value() const ;
        static os_typespec *get_os_typespec() ;
        Grade(const char *name, float value, os_segment*) ;
        ~Grade() ;

    private:
        char *name ;
        float value ;

} ;

class Course {

    public:
        int get_id() const ;
        void set_id(int) ;
        static os_typespec *get_os_typespec() ;

    private:
        int id ;

} ;
```

Main program:
dnary.cc

```
/* dnary.cc */

#include <ostore/coll/dict_pt.cc>

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

/* Student member function implementations */

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

const os_Collection<Course*> &Student::get_courses() const {
    return courses ;
}

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

    if (g) {
        grade_course.insert(g, c) ;
        course_grade.insert(c, g) ;
    } /* end if */
    return 1 ;
}

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

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

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

os_Collection<Course*> &
        Student::get_courses_with_grade(const Grade *g) const {
    os_Collection<Course*> &the_courses =
            os_Collection<Course*>::create(
            os_database::get_transient_database() ) ;
    os_cursor cur(grade_course, os_coll_range(
            os_collection::EQ, g)) ;

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

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

Student::Student(int i, os_segment *seg) :
        courses(os_Collection<Course*>::create(seg)),
        course_grade(
            os_Dictionary<const void*, Grade*>::create(seg,
            10,os_collection::pick_from_empty_returns_null) ),
        grade_course(os_Dictionary<void*, Course*>::create(seg)) {
    id = i;
}
```

```
Student::~Student() {
    os_Collection<Course*> *courses_ptr = &courses ;
    os_Dictionary<const void*, Grade*> *course_grade_ptr =
            &course_grade ;
    os_Dictionary<void*, Course*> *grade_course_ptr =
            &grade_course ;
    delete courses_ptr ;
    delete course_grade_ptr ;
    delete grade_course_ptr ;
}

/* Grade member function implementations */

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

float Grade::get_value() const {
    return value ;
}

Grade::Grade(const char *n, float v, os_segment *seg) {
    name = new( seg, os_typespec::get_char(), strlen(n)+1 )
    char[strlen(n)+1] ;
    strcpy(name, n) ;
    value = v ;
}

Grade::~Grade() {
    delete name ;
}

/* Course member function implementations */

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

void Course::set_id(int i) {
    id = i ;
}
```

Schema file:
schema.cc

```
/* schema.cc */

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

#include "dnary.hh"

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

The example explained...

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

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

The data member **Person::grade_course** contains a reference to a dictionary that maps each grade to the courses for which the student got that grade. This dictionary supports lookup of the courses given the grade.

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

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

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

If the course is not an element of the dictionary, **pick()** returns **0**, and the call to **remove()** has no effect. Note that **pick()** returns **0** in this case instead of raising an exception because the dictionary (**course_grade**) was created with **pick_from_empty_returns_null** behavior.

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

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

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

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

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

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

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

The **Student** constructor allocates an **os_Collection** and two instances of **os_Dictionary** in the specified segment. Note that the dictionary **course_grade** is created with **pick_from_empty_returns_null** behavior.

Chapter 6
Data Integrity

The information about data integrity considerations is organized in the following manner:

Data Integrity Considerations

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.

ObjectStore provides facilities to help deal with two of the most common integrity maintenance problems.

Inverse Members

One integrity control problem concerns pairs of data members that are used to model binary relationships. ObjectStore allows you to declare two data members as *inverses* of one another, so they stay in sync with each other according to the semantics of binary relationships. This works for pairs of data members that represent one-to-one, one-to-many, and many-to-many relationships. See Inverse Data Members on page 155.

Illegal Pointers

Another integrity control problem concerns *illegal pointers*. ObjectStore can detect two kinds of illegal pointers:

- Pointers from persistent memory to transient memory
- Cross-database pointers from a segment that is not in **allow_ external_pointers** mode

ObjectStore provides facilities that automatically detect such pointers upon transaction commit. You can control the way ObjectStore responds when illegal pointers are encountered; ObjectStore can either raise an exception or change the illegal pointers to **0** (null). See Detecting Illegal Pointers on page 172.

Inverse Data Members

ObjectStore allows you to model binary relationships with pointer-valued (or collection-of-pointer-valued) data members that maintain the referential integrity of their inverse data members. You implement this inverse maintenance by defining an embedded relationship class, which encapsulates the pointer (or collection-of-pointers) so that it can intercept updates to the encapsulated value, and perform the necessary inverse maintenance tasks.

The ObjectStore class library contains the necessary relationship and collection classes, as well as a set of macros to simplify the use of these classes. In general, when you use a class that has inverse members, you can access these members as if they were simple data members. The code that manipulates the instances need not be aware of the inverse maintenance that is occurring, since this is entirely hidden by the relationship class implementation.

To use ObjectStore's relationship facility, you must include the files **<ostore/relat.hh>**, along with **<ostore/ostore.hh>**, and **<ostore/coll.hh>**. The include line must place **<ostore/relat.hh>** after the other two, in the following order:

ostore/ostore.hh, ostore/coll.hh, ostore/relat.hh

Inverse Member End-User Interface

As a relationship definer (that is, the definer of the class that contains relationships), you have a number of options for presenting the relationship to that class's users. Suppose, for example, that the class **part** has a single-valued relationship **container** that points to the **part** that contains this one. Then the end user of the **part** class could be presented with any of the following interfaces for getting and setting this relationship:

Getting relationships

```
otherpart = somepart->container;   /* simple data member */
otherpart = somepart->container.getvalue();    /* relationship *
otherpart = somepart->get_container();   /*functional interface */
```

Setting relationships

```
somepart->container = otherpart;   /* simple data member */
somepart->container.setvalue(otherpart);   /* relationship */
somepart->set_container(otherpart);    /* functional interface */
```

Simple data member interface

The first style of interface is called *simple data member* because the end user interacts with the relationship exactly as if it were a simple data member of type **part***. The end user need not be aware that special *inverse-update* processing is occurring. Note, however, that this style of interface is only available in the C++ library interface, not the C library interface, because it relies on the C++ capability to define coercion operators and to overload **operator=()**.

Relationship interface

The second style of interface is called *relationship* because it treats the **container** data member as an object in its own right (that is, a relationship object). In other words, if **somepart** refers to a part, then **somepart->container** refers to a relationship instance, and **somepart->container.getvalue()** returns the value of the relationship.

Functional interface

The third style of interface is called *functional* because it encapsulates all access to the relationship inside functions defined on the class **part**.

Note that it is completely up to the class definer to decide which of these interfaces to export to the class's end users. The underlying ObjectStore library interface to relationships supports all of them and, in fact, a class definer could choose to export more than one (for example, so that the end user could do *either*

```
p->set_container(q)
```

or

```
p->container.setvalue(q))
```

Similarly, for the many-valued relationship **contents**, which lists a part's subparts, any of the following interfaces could be presented to the end user:

Getting contents

```
os_collection* subparts;
subparts = somepart->contents; /*  simple data member */
subparts = somepart->contents.getvalue(); /*  relationship */
subparts = somepart->get_contents(); /*  functional interface */
```

Setting contents

```
somepart->contents.insert(otherpart); /*  simple data member */
somepart->contents.getvalue().insert(otherpart); /* relationship */
somepart->insert_contents(otherpart); /*  functional interface */
```

Again, deciding which of these interfaces to export to the end user is under the control of the class definer. The ObjectStore library interface to relationships supports all three.

About m side of relationships

The size of an **os_relationship m** data member is eight bytes, four bytes for the pointer to the **os_collection** and four bytes for the vtbl.

The collection for an **m** side of an **os_relationship** data member is created upon the first insertion into the collection.

You control the size and placement of the collection by calling **os_relationship::create_coll()** in the constructor of the class that contains the **os_relationship m** data member.

Presizing the collection yields the best performance in terms of eliminating mutations as the collection grows, and in terms of clustering.

Defining Relationships

To define a class that has relationships, you define a data member using the appropriate relationship macro. This relationship macro defines the appropriate access functions for getting and setting the relationship. You then instantiate the bodies of these functions using another macro. Because most of the access functions have inline implementations, they incur negligible run-time overhead.

The relationship macros wrap a class around the data member; this adds no additional storage to the data member. The wrapper simply implements the functions to perform the inverse operations. The **m** side of a relationship is an embedded collection that is eight bytes. It automatically mutates to an out-of-line representation upon the insertion of the first element.

Relationship Macros

There are four relationship member macros to choose from:

- **os_relationship_1_1()** — for one-to-one relationships
- **os_relationship_1_m()** — for one-to-many relationships
- **os_relationship_m_1()** — for many-to-one relationships
- **os_relationship_m_m()** — for many-to-many relationships

The corresponding function body macros are

- **os_rel_1_1_body()** — for one-to-one relationships
- **os_rel_1_m_body()** — for one-to-many relationships
- **os_rel_m_1_body()** — for many-to-one relationships
- **os_rel_m_m_body()** — for many-to-many relationships

Descriptions of all of these macros can be found in Chapter 4, System-Supplied Macros, of the *ObjectStore C++ API Reference*.

Note that these macros always come in fours. Each use of a member macro to define one side of a relationship must be paired with another member macro to define the other side of the relationship, and each member macro must have a corresponding body macro to provide the implementations for the relationship's accessor functions. This means that a one-to-many relationship member must also have a one-to-many relationship body, *as well* *as* a many-to-one inverse member, which itself must have a many-to-one relationship body.

Macro Arguments

The member macros always have five arguments:

- Name of the class defining the member
- Name of the member
- Name of the class defining the inverse member
- Name of the inverse member
- Type signature of the member's value

Note that by scanning just the last argument and the member name, you can quickly grasp the externally visible interface to the data member. For example:

```
os_relationship_1_m (person,employer,company,employees,
        company*) employer;
```

defines a **company* employer** data member, which is part of a relationship.

The function body macros have just four arguments. For each function body macro, the arguments are exactly the same as those of the corresponding member macro, but without the last argument, as illustrated in the examples that follow.

Compiler caution

The first four macro arguments are used (among other things) to concatenate unique names for the embedded relationship class and its accessor functions. The details of macro preprocessing differ from compiler to compiler, and in some cases it is necessary to enter these macro arguments *without white space* to ensure that the argument concatenation will work correctly. There should be no white space in the argument list between the opening parenthesis and the comma separating the fourth and fifth arguments. All the examples given below follow this important convention, and should therefore work with any C++ compiler.

Relationship Examples

Example: Single-Valued Relationships

Consider an example in which a class **node** is defined that has single-valued inverse relationship members **next** and **previous** (as in a node in a list structure). This uses the **os_relationship_1_1** and **os_rel_1_1_body** macros. Note that both the simple data member and relationship style interfaces are automatically supported.

See Chapter 4, System-Supplied Macros, of the *ObjectStore C++ API Reference* for descriptions of the **os_relationship_1_1()** and **os_rel_1_1_body()** macros.

Example: **os_relationship_1_1** and **os_rel_1_1_body** macros

```
/* C++ Note Program - Header File */

#include <fstream.h>
#include <string.h>
#include <ostore/ostore.hh>
#include <ostore/coll.hh>
#include <ostore/relat.hh>

class author;

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

    public:

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

        /* Public Data members */
        os_backptr bkptr;
        char* user_text;
        os_indexable_member(note,priority,int) priority;
        os_relationship_1_m(
            note,the_author,author,notes,author*)
            the_author;
};

#include <ostore/relat.hh>

class node {

    public:

        os_relationship_1_1(node,next,node,previous,node*) next;
        os_relationship_1_1(node,previous,node,next,
            node*) previous;
        node() {};
};

os_rel_1_1_body(node,next,node,previous);
os_rel_1_1_body(node,previous,node,next);

main() {
    /* show the end users use of these relationships */

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

    node*  n1 = new node();
    node*  n2 = new node();

    n1->next = n2;

    /* this also automatically updates n2->previous  */
    printf("n1 (%x) --> (%x)\n",
        n1, n1->next.getvalue());

    printf("n2 (%x) --> (%x)\n",
        n2, n2->previous.getvalue());

}
```

Compiler caution

While the simple data member style of access normally allows you to treat a single-valued relationship as a normal pointer-valued data member in most situations, this capability depends upon the **operator=()** (to set the value) and coercion operators (to get the value). Thus, the following simple assignment,

```
n1->next = n2->next;
```

actually is interpreted by the C++ compiler as

```
n1->next.operator=( n2->next.operator node* () );
```

Example: incorrect use of the coercion operator

The coercion operator **operator node* ()** is used to get the value of the relationship in the right-hand-side expression, and the assignment operator **operator=()** is used to set the value of the relationship in the left-hand-side expression. Be aware that the compiler will only apply the coercion operator if it knows that the desired type of the expression is a **node*** pointer. The following *will not work correctly*:

```
printf("The value of the relationship is %x \n", n1->next );
```

because **printf()** does not have prototype information for its arguments, so the compiler does not know to apply a coercion. In this case, either of the following would be a suitable alternative:

Example: avoiding coercion errors

```
printf("The value of the relationship is %x \n",
        n1->next.getvalue() );

printf("The value of the relationship is %x \n",
        (node*)n1->next );
```

Example: private declarations of relationships

The next example defines a class **node** just as above, but presents to the end user a functional style interface. This is done exactly as above, except that the relationships themselves are declared private, so that the user cannot directly access them via the simple data member or relationship-style interfaces; and the class-definer writes simple inline member functions to extend a functional-style interface instead. Note that in this example the two relationship members are defined by the same class, **node**. This would not have to be the case. Even if they were defined by different classes, say **node** and **arc**, they could still be made private, because the relationship macros define the relationship implementation classes as friends.

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

class node {
    private:
        os_relationship_1_1(node,next,node,previous,
            node*) next;
        os_relationship_1_1(node,previous,node,next,
            node*) previous;
    public:
        node* get_next() {return next.getvalue();};
        void set_next(node* val) {next.setvalue(val);};

        node* get_previous() {
        return previous.getvalue();};
        void set_previous(node* val) {
        previous.setvalue(val);};
        node() {};
};

os_rel_1_1_body(node,next,node,previous);
os_rel_1_1_body(node,previous,node,next);

main() {

/* show the end users use of these relationships */

    objectstore::initialize();
    os_collection::initialize();
    node*  n1 = new node();
    node*  n2 = new node();
```

```
        n1->set_next(n2);
        /* this automatically also updates n2->prev */

        printf("n1 (%x) --> (%x)\n",n1, n1->get_next());
        printf("n2 (%x) --> (%x)\n",n2, n2->get_prev());
    }
```

Example: Many-Valued Relationships

The **os_rel_m_m_body** and **os_rel_m_1_body** macros should not be used in include files that are included in more than one source file used in a given application. This is because these macros define the bodies for virtual functions. Using these macros in a header file that is included in more than one place can result in redundant definitions of the virtual table that is generated by the compiler to implement virtual function calling.

See Chapter 4, System-Supplied Macros, of the *ObjectStore C++ API Reference* for descriptions of the **os_rel_m_m_body()**, **os_rel_m_1_body()**, and **os_relationship_m_m()** macros.

Example: **os_relationship_m_m** and **os_rel_m_m_body** macros

Here is an example in which a class **node** is defined with a pair of many-to-many relationships, **ancestors** and **descendents** (as in a node in a graph structure).

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

class node {
    public:
        os_relationship_m_m(node,ancestors,node,descendents,
            os_collection) ancestors;
        os_relationship_m_m(node,descendents,node,ancestors,
            os_collection) descendents;
        node() {};
};

os_rel_m_m_body(node,ancestors,node,descendents);
os_rel_m_m_body(node,descendents,node,ancestors);

main() {

    /* show the end users use of these relationships */
    objectstore::initialize(); os_collection::initialize();
    node*  n1 = new node(); node*  n2 = new node();

    n1->ancestors.insert(n2);
    /* this also updates n2->descendents */

    node* n;

    printf("n1 (%x)\n",n1);
printf(" has %d descendents: ", n1->descendents->size ()); {
    os_cursor c(n1->descendents);
    for (n = (node*) c.first(); n; n = (node*) c.next())
        printf("(%x) ",n);
    printf("\n");
}

printf("    and %d ancestors: ", n1->ancestors->size ()) {
    os_cursor c(n1->ancestors);
    for (n = (node*) c.first(); n; n = (node*) c.next())
        printf("(%x) ", n);
    printf("\n");
}

printf("n2 (%x)\n",n2);
printf("    has %d descendents: ",
    n2->descendents->size ()); {
    os_cursor c(n2->descendents);
    for (n = (node*) c.first(); n; n = (node*) c.next())
        printf("(%x) ", n);
    printf("\n");
}

printf("    and %d ancestors: ",
    n2->ancestors->size ()); {
    os_cursor c(n2->ancestors);
    for (n = (node*) c.first(); n; n = (node*) c.next())
        printf("(%x) ", n);
    printf("\n");
}

}
```

Example: One-to-Many and Many-to-One Relationships

Below is an example in which a class **node** is defined that has a one-to-many relationship, **children**, and a many-to-one inverse, **parent** (as in a node in a tree structure).

See Chapter 4, System-Supplied Macros, of the *ObjectStore C++ API Reference* for descriptions of the **os_relationship_1_m()**, **os_relationship_m_1()**, **os_rel_1_m_body()**, and **os_rel_m_1_body()** macros.

Example: **os_relationship_1_m**, **os_relationship_m_1**, **os_rel_1_m_body**, and **os_rel_m_1_body** macros

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

class node {

    public:
        os_relationship_1_m(node,parent,node,children,
            node*) parent;
        os_relationship_m_1(node,children,node,parent,
            os_collection) children;
        node() {};
};

os_rel_1_m_body(node,parent,node,children);
os_rel_m_1_body(node,children,node,parent);

main() {

    /* show the end users use of these relationships */

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

    node*  n1 = new node();
    node*  n2 = new node();

    n1->children.insert(n2);
    /* this also updates n2->parent */
    /* NOTE: "n2->parent = n1;" would have had */
    /* identical effect */

    /* etc */
}
```

Example: one-to-many with different classes

Here is an example that illustrates a one-to-many relationship involving two different classes.

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

class person {
    public:
        os_relationship_1_m(person,employer,company,
            employees, company*) employer;
        char* name;
};

class company {
    public:
        os_relationship_m_1(company,employees,person,
            employer, os_collection) employees;
    int gross_revenue;
};

os_rel_1_m_body(person,employer,company,employees);
os_rel_m_1_body(company,employees,person,employer);
```

Duplicates and Many-Valued Inverse Relationships

For most kinds of ObjectStore relationships, an update to one side of the relationship always triggers a corresponding update to the other side. This is true for the following kinds of relationships:

- One-to-one relationships
- One-to-many relationships, where the collection involved does not allow duplicates
- Many-to-many relationships, where the collections involved either both allow duplicates or both disallow duplicates

For other relationships, an update to one side does not always trigger an update to the other side.

Example: one-to-many relationship when duplicates are allowed

The following example shows how ObjectStore handles one-to-many relationships where the collection at the *many* end of the relationship allows duplicates. It also shows how ObjectStore handles many-to-many relationships where one of the collections involved allows duplicates and the other does not.

Suppose a complex part keeps track of the primitive parts it uses, as well as how many times each primitive part is used. (For example, a wheel might be a primitive part, and be used four times in a complex part like a car.) Suppose also that each primitive part is used in only one complex part. This can be modeled with the following classes:

Class definitions

```
class complex_part {
    os_relationship_m_1(
            complex_part,
            components,
            primitive_part,
            used_by,
            os_Bag<primitive_part*> ) components ;
}

class primitive_part {
    os_relationship_1_m(
            primitive_part,
            used_by,
            complex_part,
            components,
            complex_part* ) used_by ;
}
```

Suppose that a certain **primitive_part**, **a_wheel**, is used by a particular **complex_part**, **the_car**. If you do

```
a_wheel->used_by = 0;
```

ObjectStore removes all occurrences of **a_wheel** from **the_car**'s components, since setting **used_by** to **0** implies that the wheel is not used by the car at all.

Suppose you do

```
the_car->components.remove(a_wheel)
```

If the car uses four wheels at first, afterward it uses three wheels. **a_wheel->used_by** still points to the car, since the car still uses the wheel at least once.

Now suppose each primitive part can be used by multiple complex parts.

```
class complex_part {
    os_relationship_m_1(
            complex_part,
            components,
            primitive_part,
            used_by,
            os_Bag<primitive_part*>
    ) components ;
}
class primitive_part {
    os_relationship_1_m(
            primitive_part,
            used_by,
            complex_part,
            components,
            os_Set<complex_part*>
    ) used_by ;
}
```

And suppose you do

```
a_wheel->used_by.remove(the_car);
```

This causes all occurrences of **a_wheel** to be removed from **the_car**'s components, since it implies that the wheel is not used by the car at all.

If you do

```
the_car->components.remove(a_wheel);
```

ObjectStore removes **the_car** from the wheel's **used_by** set only if it removes the last occurrence of the wheel from the car's components, that is, only if the car no longer uses the wheel at all.

Use of Parameterized Types

Relationships can be used either with or without a compiler that supports parameterized types. All the previous examples were written without the use of parameterization. In the case of many-valued relationships, a greater degree of type safety can be obtained by using a parameterized collection type. This is accomplished by changing the last parameter to the relationship member macro (recall that the last parameter always indicates the type of the value). For example:

```
class node {
    public:
        os_relationship_m_m(node,ancestors,node,descendents,
            os_Collection<node*>) ancestors;
        os_relationship_m_m(
                node,descendents,node,ancestors,
                os_Collection<node*>) descendents;
                node() {};
};

os_rel_m_m_body(node,ancestors,node,descendents);
os_rel_m_m_body(node,descendents,node,ancestors);
```

In this case, the functions that perform a get-value (that is, **getvalue()**), and the coercion operator will return an **os_ Collection<node*>&** rather than just an **os_collection&**.

Deletion Propagation and Required Relationships

By default, deleting an object that participates in a relationship automatically updates the other side of the relationship so that there are no dangling pointers to the deleted object. In some cases, however, the desired behavior is actually to delete the object on the other side of the relationship (for example, for subsidiary component objects). You can obtain this behavior by using the relationship body macros:

- **os_rel_1_1_body_options()**
- **os_rel_1_m_body_options()**
- **os_rel_m_1_body_options()**
- **os_rel_m_m_body_options()**

(Descriptions of all of these macros can be found in Chapter 4, System-Supplied Macros, of the *ObjectStore C++ API Reference*.)

These macros are like the body macros already discussed, except that they have three extra arguments, used for specifying various options. The fifth argument (the first extra argument) can be either **os_rel_propagate_delete** or **os_rel_dont_propagate_delete**, as in

Example

```
os_rel_m_1_body_options(part,subparts,part,container,
        os_rel_propagate_delete, os_auto_index, os_no_index)
```

The last two arguments are used to indicate whether the current member and its inverse are indexable. These are described in the next section.

Indexable Inverse Members

If you want automatic index maintenance enabled for an inverse data member, you must use one of the options body macros:

- **os_rel_1_1_body_options()**
- **os_rel_1_m_body_options()**
- **os_rel_m_1_body_options()**
- **os_rel_m_m_body_options()**

(Descriptions of all these macros can be found in Chapter 4, System-Supplied Macros, of the *ObjectStore C++ API Reference*.)

These macros are like the body macros discussed earlier, except that they have three extra arguments, used for specifying various options.

The sixth and seventh arguments (the second and third extra arguments) are used to specify whether the current member and its inverse are indexable, respectively. For nonindexable members, use **os_no_index**. For indexable members, use a call to the macro **os_index(),** indicating the name of the defining class's **os_backptr** member. Such macro calls have the form

Form of the call

os_index(*class*,*member*)

where *class* is the name of the class defining the indexable member, and *member* is the name of the **os_backptr-**valued data member appearing before indexable members of the class. Here is an example:

Example

```
os_rel_m_1_body_options(part,subparts,part,container,
    os_propagate_delete,
    os_auto_index, os_index(part,b))
```

Many-valued members that have an inverse do not need to be indexable to be used in a path. For an indexable many-valued relationship, specify **os_auto_index**.

Detecting Illegal Pointers

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 (see Referring Across Databases and Transactions on page 61). If you subsequently retrieve and dereference an illegal pointer, you most likely will get an exception.

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. Keep in mind that this can be expensive and should really be done for development and stress testing, but not in production. You can also direct ObjectStore *never* to check. See Controlling Illegal Pointer Checking on page 173.

You can also control the action taken when ObjectStore detects an illegal pointer, as described in Controlling the Consequences of Illegal Pointer Detection on page 175.

Note that these functions concern checking for pointers to transient memory and cross-database pointers. There are other kinds of database pointers that can cause database integrity problems. Prominent among these is the *dangling reference*, the pointer to a deleted object, as well as the incorrectly typed pointer. The integrity functions described in the following sections do not involve checking for such pointers. However the inverse member and schema evolution facilities do provide integrity control support in these areas.

Controlling Illegal Pointer Checking

You control whether ObjectStore always detects illegal pointers with the following member of **os_segment**:

```
void set_check_illegal_pointers(os_boolean) ;
```

Setting behavior for checking for illegal pointers

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) to **set_check_illegal_pointers()**. The function's formal parameter is **os_boolean**, which is defined as **int** or **long**, whichever is 32 bits on your platform.

The results of using this function do not remain in effect after the current process ends, and are invisible to other processes. See Illegal Pointer Modes Are Process Local on page 177.

Finding illegal pointers in a given segment

You can use the following member of **os_segment** to determine the current mode of a given segment:

```
os_boolean get_check_illegal_pointers();
```

If the segment is in **check_illegal_pointers** mode, the function returns **1**; otherwise, it returns **0**.

Creating new segments with checking enabled

For a given database, you can direct ObjectStore to create new segments in **check_illegal_pointers** mode by passing **1** to the following member of **os_database**:

```
void set_default_check_illegal_pointers(os_boolean) ;
```

Finding current mode of a database

You can determine if a database is in **default_check_illegal_pointers** mode with the following member of **os_database**:

```
os_boolean get_default_check_illegal_pointers() ;
```

Enabling pointer checking by segment or database

For a given database, you can enable **default_check_illegal_pointers** mode and enable **check_illegal_pointers** mode for each segment in the database by passing **1** to the following member of **os_database**:

```
void set_check_illegal_pointers(os_boolean) ;
```

Passing **0** disables **default_check_illegal_pointers** mode for the database and disables **check_illegal_pointers** mode for each segment in the database.

Making pointer checking default behavior for new databases

You can direct ObjectStore to create new databases in **default_check_illegal_pointers** mode by passing **1** to the following member of the class **objectstore**:

```
static void set_check_illegal_pointers(os_boolean) ;
```

This also enables **check_illegal_pointers** mode for each database currently retrieved by the current process.

Finding default behavior of current process

You can determine if the current process enables **default_check_illegal_pointers** for newly created databases with the following member of the class **objectstore**:

```
static os_boolean get_check_illegal_pointers() ;
```

Prevent checking for illegal pointers

You can direct ObjectStore never to check for illegal pointers with the following member of the class **objectstore**:

```
static void always_ignore_illegal_pointers(os_boolean) ;
```

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.

Determining which class object is triggering an error

If you are getting an exception about an illegal pointer during transaction commit and it is unclear which class object the exception is being signaled for, you might force the illegal pointer to be written to the database by ignoring illegal pointers. Then a subsequent **osverifydb** operation of the database should tell you which class / data member has the illegal pointer. With this information you can inspect your code for all functions that set this data member, to see if an application error is present.

Controlling the Consequences of Illegal Pointer Detection

You control what happens when ObjectStore finds an illegal pointer with the following member of **os_segment**:

```
void set_null_illegal_pointers(os_boolean) ;
```

Default error signaling for illegal pointers

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 Illegal Pointer Modes Are Process Local on page 177.

Find the current signal mode of a given segment

You can use the following member of **os_segment** to determine the current mode of a given segment:

```
os_boolean get_null_illegal_pointers() ;
```

If the segment is in **null_illegal_pointers** mode, the function returns nonzero; otherwise, it returns **0**.

Making illegal pointers null the default behavior

For a given database, you can direct ObjectStore to create new segments in **null_illegal_pointers** mode by passing **1** to the following member of **os_database**:

```
void set_default_null_illegal_pointers(os_boolean) ;
```

Finding default behavior of current database

You can determine if a database is in **default_null_illegal_pointers** mode with the following member of **os_database**:

```
os_boolean get_default_null_illegal_pointers() ;
```

Making illegal pointers null for a specific database

For a given database, you can enable **default_null_illegal_pointers** mode, and enable **null_illegal_pointers** mode for each segment in the database, by passing **1** to the following member of **os_ database**:

```
void set_null_illegal_pointers(os_boolean) ;
```

Passing **0** disables **default_null_illegal_pointers** mode for the database and disables **null_illegal_pointers** mode for each segment in the database.

Making nullification of illegal pointers the default behavior for new databases

You can direct ObjectStore to create new databases in **default_null_ illegal_pointers** mode by passing **1** to the following member of the class **objectstore**:

static void set_null_illegal_pointers(os_boolean) ;

This also enables **null_illegal_pointers** mode for each database currently retrieved by the current process.

Finding default behavior of current process

You can determine if the current process enables **default_null_ illegal_pointers** for newly created databases with the following member of the class **objectstore**:

static os_boolean get_null_illegal_pointers() ;

Setting behavior to ignore illegal pointers

You can direct ObjectStore to ignore illegal pointers whenever detected, instead of signaling an exception, by using the following member of the class **objectstore**:

static void set_always_ignore_illegal_pointers(os_boolean) ;

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.

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.

Illegal Pointer Modes Are Process Local

The modes relating to illegal pointers are all process local. The results of using the functions described here do not remain in effect after the current process terminates, and the results are invisible to other processes.

For example, if one process enables **null_illegal_pointers** mode for a given segment, another concurrent process can disable **null_ illegal_pointers** mode for that same segment. The first process will change illegal pointers to **0**, and the second process will signal an error when it finds an illegal pointer. Moreover, the modes are transient; they remain in effect only until the process terminates. So these modes actually determine the nature of illegal pointer checking and handling *for the current process*.

Chapter 7
Database Access Control

ObjectStore offers a choice of methods for controlling who can access a database. The information describing techniques you can use is organized in the following manner:

Access Control Methods

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. See Chapter 2, Server Parameters, of *ObjectStore Management* for more information about access control methods.

Setting User Category Permissions

For rawfs databases, you can specify a combination of types of access to a given directory, database, or segment for a given category of users.

To specify access restrictions on a rawfs database or directory, use **os_dbutil::chmod()** or the utility **oschmod**. To specify access restrictions on a segment in a rawfs database, use **os_segment::set_access_control()**. See Chapter 2, Class Library, of the *ObjectStore C++ API Reference*, as well as Chapter 4, Utilities, of *ObjectStore Management*.

For file databases, you can specify a combination of types of access to a given directory or database for a given category of users. To do this, use commands of the native operating system. With file databases, a segment always has the same protections as the database that contains it; see the discussions beginning with Categories of Users on page 181.

Restricting Database Access Using Schema Keys

If you want to restrict access to a database's data and *metadata*, use ObjectStore's schema protection facility. This facility allows you to associate a *schema key* (a pair of integers) with a database. Once a database has been given a schema key, an application must supply the key in order to access data in the database. See the discussions beginning with Schema Keys on page 192.

Categories of Users

For a given directory, database, or segment there are three categories of users:

- Its *owner*
- Users in its *primary group*
- Those in its *default group*, that is, everyone else

Owner of a Directory, Database, or Segment

For file databases and the directories they occupy, the owner is determined by the native operating system and its commands. For rawfs databases and the directories they occupy, the owner is initially the creator, and is subsequently determined by **os_ dbutil::chown()** and the utility **oschown**. The owner of a segment is the owner of the database that contains it.

Only the owner of a segment can modify its protections with **os_ segment::set_access_control()**.

Group of a Directory, Database, or Segment

For file databases, the primary group of a directory or database is determined by the native operating system and its commands. For rawfs databases, the primary group of a directory or database is initially the owner's group, and is subsequently determined by **os_dbutil::chgrp()** and the utility **oschgrp**. The primary group of a segment is initially the group of the containing database, and is subsequently determined by **os_segment::set_access_control()**.

Group of a User

A user's group membership is determined by the native operating system and its commands.

Permissions

For rawfs databases, you can specify access permissions at three levels of granularity: directory, database, and segment.

For file databases, you can specify access permissions at two levels of granularity: directory and database.

Directory Permissions

At any time, each user has zero or more of the following two types of access to a given directory:

- Write: allows the user to create and delete databases and directories contained in it. For file databases, necessary for the user to update databases it contains.
- Read: allows the user to list its content. For file databases, necessary for the user to read databases it contains. Note that rawfs directories do not follow the UNIX convention in distinguishing read and execute access.

Database Permissions

At any time, each user has zero or more of the following two types of access to a given database:

- Write: necessary for the user to open the database for update, and, for file databases, to create and modify its data and metadata
- Read: necessary for the user to open the database for read, and, for file databases, to read the data and metadata it contains

Segment Permissions

For a given segment in a rawfs database, each user has zero or more of the following two types of access to the segment:

- Write: necessary for the user to update the data it contains
- Read: necessary for the user to read the data it contains

Each new rawfs segment grants both read and write permissions to all three categories of users.

Permission Checks

Directory-Level Access

To create or delete databases in a given directory, you must have write access to the directory. To list the contents of a directory, you must have read access to the directory.

Database-Level Access

For an application to open a database for update, the user that launched the application must have write permission for the database. For file databases, the user must also have write permission for the directory containing the database.

For an application to open a database for read, the user must have read permission for the database. For file databases, the user must also have read permission for the directory containing the database.

If a user does not have the appropriate permissions when opening a database for read or update, ObjectStore signals err_permission_denied.

Segment-Level Access

Consider a segment in a rawfs database. In each transaction, the first time an application attempts to lock (for read or write) data in the segment, ObjectStore checks the access permissions granted to the user that launched the application.

For the application to perform write access on the segment, the user must have write permission for the segment. For file databases, the user must also have write permission for the directory containing the database that contains the segment, as well as write permission for the database.

For the application to perform read access on the segment, the user must have read permission for the segment. For file databases, the user must also have read permission for the directory containing the database that contains the segment, as well as read permission for the database.

If a user does not have the appropriate permissions when attempting initial read or write access to a segment, ObjectStore signals err_permission_denied.

Segment-Level Permissions API

The programming interface for rawfs database segment-level permissions is provided by the following functions:

- Members of the class **os_segment**
- **os_segment::set_access_control()**
- **os_segment::get_access_control()**
- **os_database::get_all_segments_and_permissions()**

These and other ObjectStore functions are described in detail in Chapter 2, Class Library, of *ObjectStore Management*.

Establishing Access Permissions with 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 (see Categories of Users on page 181).

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.

Access type enumerators

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.

For more information, see **os_segment_access** in Chapter 2 of the *ObjectStore C++ API Reference*.

os_segment_access::set_primary_group()

There are two overloadings of this function. The first is declared as follows:

```
void set_primary_group(
    const char* group_name,
    os_int32 access_type
) ;
```

This function 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**.

The second overloading is declared as follows:

```
void set_primary_group(
    os_int32 access_type
    );
```

This function 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**.

For more information, see **os_segment_access::set_primary_group()** in Chapter 2 of the *ObjectStore C++ API Reference*.

os_segment_access::get_primary_group()

This function is declared as follows:

```
os_int32 get_primary_group(
        const char** group_name = 0
) const ;
```

It 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.

For more information, see **os_segment_access::get_primary_group()** in Chapter 2 of the *ObjectStore C++ API Reference*.

os_segment_access::set_default()

This function is declared as follows:

```
void set_default(
        os_int32 access_type
);
```

This function 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**.

For more information, see **os_segment_access::set_default()** in Chapter 2 of the *ObjectStore C++ API Reference*.

os_segment_access::get_default()

This function is declared as follows:

```
os_int32 get_default() const ;
```

It returns the types of access associated with the default group for the **os_segment_access**.

For more information, see **os_segment_access::get_default()** in Chapter 2 of the *ObjectStore C++ API Reference*.

os_segment_access::os_segment_access()

This function has three overloadings. The first is declared as follows:

```
os_segment_access() ;
```

This creates an **os_segment_access** that associates **no_access** with both the default group and the group named **group_name**.

The second overloading is declared as follows:

```
os_segment_access(
    const char* primary_group,
    os_int32 primary_group_access_type,
    os_int32 default_access_type
) ;
```

This 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**.

The third overloading is declared as follows:

```
os_segment_access(
     const os_segment_access& source
) ;
```

This 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.

For more information, see **os_segment_access::os_segment_access()** in Chapter 2 of the *ObjectStore C++ API Reference*.

os_segment_access::operator =()

This function is declared as follows:

```
os_segment_access& operator= (
    const os_segment_access& source
) ;
```

This function 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**.

For more information, see **os_segment_access::operator =()** in Chapter 2 of the *ObjectStore C++ API Reference*.

os_segment_access::~os_segment_access()

The destructor frees memory associated with the deleted instance of **os_segment_access**.

For more information, see **os_segment_access::~os_segment_access()** in Chapter 2 of the *ObjectStore C++ API Reference*.

os_segment::set_access_control()

This function is declared as follows:

```
void set_access_control( const os_segment_access
         *new_access) ;
```

This associates the specified **os_segment_access** with the specified segment. The **os_segment_access** determines the primary group and permissions for the **os_segment**. 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()**.

For more information, see **os_segment::set_access_control()** in Chapter 2 of the *ObjectStore C++ API Reference*.

os_segment::get_access_control()

This function is declared as follows:

```
os_segment_access *get_access_control() const ;
```

It returns a pointer to the segment's associated **os_segment_access**, which indicates the segment's primary group and permissions.

For more information, see **os_segment::get_access_control()** in Chapter 2 of the *ObjectStore C++ API Reference*.

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** segs,
    os_segment_access** 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.

For more information, see **os_database::get_all_segments_and_permissions()** in Chapter 2 of the *ObjectStore C++ API Reference*.

Segment-Level Permissions and Locking

When you change a segment's permissions, ObjectStore locks the entire segment for write. This means that permission changes at the segment level are transaction consistent (database-level changes are not transaction consistent). But this can also adversely affect the performance of the application performing the change, as well as of concurrent applications, if there is a lot of contention for the segment's pages.

Permissions and Related Segments

Remember to set the permissions on related segments in a compatible way. That is, if an operation on one segment triggers access to another segment, do not forget that *both* segments must have the appropriate permissions.

For example, consider a collection in one segment that has an index in another segment. If you perform a query on the collection that uses the index, do not forget that both segments must be accessible for read.

Similarly, given a configuration that spans segments, unexpected permission failures can occur if all the configuration's segments do not have the same protections.

Schema Keys

If you want to restrict access to a database's data and *metadata*, use ObjectStore's schema protection facility. This facility allows you to associate a *schema key* (a pair of integers) with a database. Once a database has been given a schema key, an application must supply the key in order to access data in the database.

Database Schema Keys

At any time a database can have a schema key that consists of a pair of four-byte unsigned integers. By default, a database has no schema key. You specify a key for a database with **os_database::change_schema_key()**. You can also freeze the schema key of a given database, preventing any change to the key, even by applications with a matching key. You do this with **os_database::freeze_schema_key()**. See Schema Key API on page 194.

If a database has a schema key, it can only be accessed by an application that supplies a matching schema key. See Application Schema Keys on page 192 as well as Key Mismatch on page 193.

Application Schema Keys

As is the case with ObjectStore databases, ObjectStore applications can have a schema key that consists of a pair of four-byte unsigned integers. By default, an application has no current schema key. You specify a key for an application with **objectstore::set_current_schema_key()**. See Schema Key API on page 194. For ObjectStore tools and for ObjectStore utilities executed from the command line, you specify the schema key with environment variables. See Schema Key Environment Variables on page 197.

Key Mismatch

When an application first attempts to access the data or metadata of a protected database (one with a schema key), an err_schema_key is signaled if the application has no key or has a different key from the database. ObjectStore issues an error message like the following:

```
Error using schema keys
<err-0025-0151>The schema is protected and the key, if provided,
did not match the one in the schema of database db1.
(err_schema_key)
```

If the database's open count goes to 0 and then is increased to 1 again, the schema protection key is checked again at the first attempted access. In general, the schema protection key is checked at the first attempted access after each database open that increments the open count from 0 to 1.

If an application is linked with a version of ObjectStore that does not support schema protection, and the application tries to access a database with a schema key, err_uninitialized is signaled. ObjectStore issues an error message like the following:

```
The database is uninitialized.
<err-0025-0508>The database db1 is corrupted or uninitialized.
Possibly the transaction that created this database aborted.
Use osrm to remove the database, and try again. (err_uninitialized)
```

Permissible operations on a protected database

You can perform certain operations on a database that are not considered to access the database, since they do not require use of the database's schema. These operations include opening and copying a database. You can perform these operations on a protected database without knowing its key.

Caution when using the **oscp** utility

In the case of the **oscp** utility , however, specifying the correct key can nevertheless affect the operation's result. If you specify the correct key, the copy has a different **db_id** from the original; otherwise, the copy has the same **db_id** as the original. In either case, the copy has the same key as the original, and the copy's key is frozen if and only if the original's is. For more information on the **oscp** utility, see oscp: Copying Databases in Chapter 4 of *ObjectStore Management*.

Schema Key API

The programming interface for schema protection is provided by the following functions:

- **os_database::change_schema_key()**
- **objectstore::set_current_schema_key()**
- **os_database::freeze_schema_key()**

Setting a Database Schema Key with change_schema_key()

You set the schema key of a database with **os_database::change_schema_key()**. This function is declared as follows:

```
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
) ;
```

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.
```

old_key_low and **old_key_high**

If the database already has a schema key at the time of the call, **old_key_low** must be the first component of the key and **old_key_high** must be 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 and **new_key_high**

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.

For more information, see **os_database::change_schema_key()** in Chapter 2 of the *ObjectStore C++ API Reference*.

Setting Application Schema Keys with set_current_schema_key()

The function **objectstore::set_current_schema_key()** can be used to set or unset the schema key of the current application. This function is declared as follows:

```
void objectstore::set_current_schema_key(
    os_unsigned_int32 key_low,
    os_unsigned_int32 key_high
) ;
```

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 and **key_high**

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 **OS_SCHEMA_KEY_HIGH** and **OS_SCHEMA_KEY_LOW** (see Schema Key Environment Variables on page 197). If neither variable is set, the application has no current schema key.

For more information, see **objectstore::set_current_schema_key()** in Chapter 2 of the *ObjectStore C++ API Reference*.

Freezing a Database Key with freeze_schema_key()

Use **os_database::freeze_schema_key()** to freeze a database's key, preventing any change to the key, even by applications with a matching key. This function is declared as follows:

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

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-0025-0159>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.
```

key_low and **key_high**

key_low and **key_high** must also match the database's schema key, or else err_schema_key is signaled.

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

For more information, see **os_database::freeze_schema_key()** in Chapter 2 of the *ObjectStore C++ API Reference*.

Schema Key Environment Variables

If you run certain ObjectStore tools and utilities on schema-protected databases, set the ObjectStore client environment variables **OS_SCHEMA_KEY_LOW** and **OS_SCHEMA_KEY_HIGH** to specify the schema key of the databases to be accessed.

Normally, you specify a key for an application with **objectstore::set_current_schema_key()**. These environment variables are provided since it is not possible for you to set the schema key of a tool or utility programmatically. ObjectStore environment variables are described in Chapter 3, Environment Variables, of *ObjectStore Management*.

The tools and utilities for which these variables must be set include the following:

Utility	***Function***	***Refer to Chapter 4, Utilities, in ObjectStore Management***
oscompact	Removes deleted space in specified databases or segments.	oscompact: Compacting Databases
osexschm	Lists the names of all classes in the schema referenced by the specified database.	osexschm: Displaying Class Names in a Schema
ossevol	Modifies a database and its schema so that it matches a revised application schema.	ossevol: Evolving Schemas (also Schema Evolution with ossevol on page 204 of this publication)
ossg	ObjectStore schema generator.	ossg: Generating Schemas
ossize	Displays the size of the specified database and the sizes of its segments.	ossize: Displaying Database Size
osverifydb	Verifies all pointers and references in a database.	osverifydb: Verifying Pointers and References in a Database (also Using osverifydb to Verify Pointers and References on page 206 of this publication)

These environment variables determine an application's schema key when an ObjectStore application attempts to access data in a schema-protected database, and either one of the following is true:

- The application did not set the schema key using **objectstore::set_current_schema_key()**.
- The application's most recent call to **object-store::set_current_ schema_key()** specified 0 for both arguments.

Keep in mind that when the environment variables determine an application's schema key, all schema-protected databases that the application accesses must have the same schema key.

By default, neither **OS_SCHEMA_KEY_LOW** nor **OS_SCHEMA_ KEY_HIGH** is set.

Building applications that allow utility use on protected databases

To allow your customers to use an ObjectStore utility on a database that you have protected, build an application that calls the member of the class **os_dbutil** that corresponds to the utility. This application can specify the schema key with **objectstore::set_ current_schema_key()** (see Chapter 2 in *ObjectStore C++ API Reference*).

Some ObjectStore tools (such as the **ossg** utility) cannot be invoked from the ObjectStore API. To allow your customers to use such a tool on a database that you have protected, build an application that spawns the tool as a child process, and specify the key of the child process by setting the environment variables from within the application.

Chapter 8
Schema Evolution

This chapter provides an introduction to ObjectStore schema evolution. Schema evolution is a complex facility; only the most basic schema evolution operations are discussed in this chapter. Detailed coverage of this topic is found in Chapter 9, Advanced Schema Evolution, in the *ObjectStore Advanced C++ API User Guide*.

The material is organized in the following manner:

What Is 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 representations of objects already stored in the database.

ObjectStore provides some basic utilities to use in uncomplicated circumstances, as well as a schema evolution interface that lets you write a specialized schema evolution application. With these tools, you can redefine the classes in a database's schema to accommodate changes. ObjectStore can modify the schema and change the representations of any existing instances in the database to conform to the new class definitions.

Without the the schema evolution facility, you can only change a database schema by adding new classes to it. You cannot redefine a class already contained in the schema, except in ways that do not affect the layout that the class defines for its instances. For example, adding a nonstatic data member changes instance layout, but adding a nonvirtual member function does not.

Making Use of Schema Evolution

It is difficult to predict exactly when you might need to redefine a class or classes in a database. Schema evolution is a complex operation. If it requires that you write a special schema evolution application, the process must be planned and executed very methodically with many checkpoints along the way. Schema evolution is a very powerful tool. It allows you to change the semantics of the objects in your database. As such, it poses the very real danger of user-introduced data corruption.

Planning Your Schema Evolution

As you develop and implement your schema evolution plan, you must ensure that

- No unanticipated results affected the data.
- The evolution was complete.

When planning your schema evolution, it is essential to anticipate all your requirements for the evolution, and to plan your application carefully around the desired outcome. A good rule of thumb is to try your schema evolution application on small databases to investigate the process, determine what works, and locate anything that might introduce complications.

Planning in the development phase of schema evolution is critical, but of equal importance are careful testing and validation of your implementation using a variety of methods. A conservative approach is best, so plan for the stages of your schema evolution project in the following sequence.

Restriction note

When planning a schema evolution, be aware that if the database is greater than *n*% of **OS_AS_SIZE** (the environment variable establishing the size of the client's persistent storage space), schema evolution does not work. The value of *n*% varies greatly depending on the specifics of the applications involved. Consult Object Design Technical Support for advice about the amount of persistent storage space required by schema evolution for your particular configuration.

Regardless of the method used to perform schema evolution, before evolution starts, use the **osverifydb -all** utility to check for database errors. (See Using osverifydb to Verify Pointers and References on page 206.)

Sequence of Planning Your Schema Evolution

1 Determine if you can use the **ossevol** utility to update a database’s schema, or if you must design a special schema evolution application. You cannot, for example, use **ossevol** to update a database if the database contains instances of **os_Dictionary** or **os_rDictionary**. (See Schema Evolution with ossevol on page 204 for more information).

2 Plan the schema evolution model in cases where you require a special application.

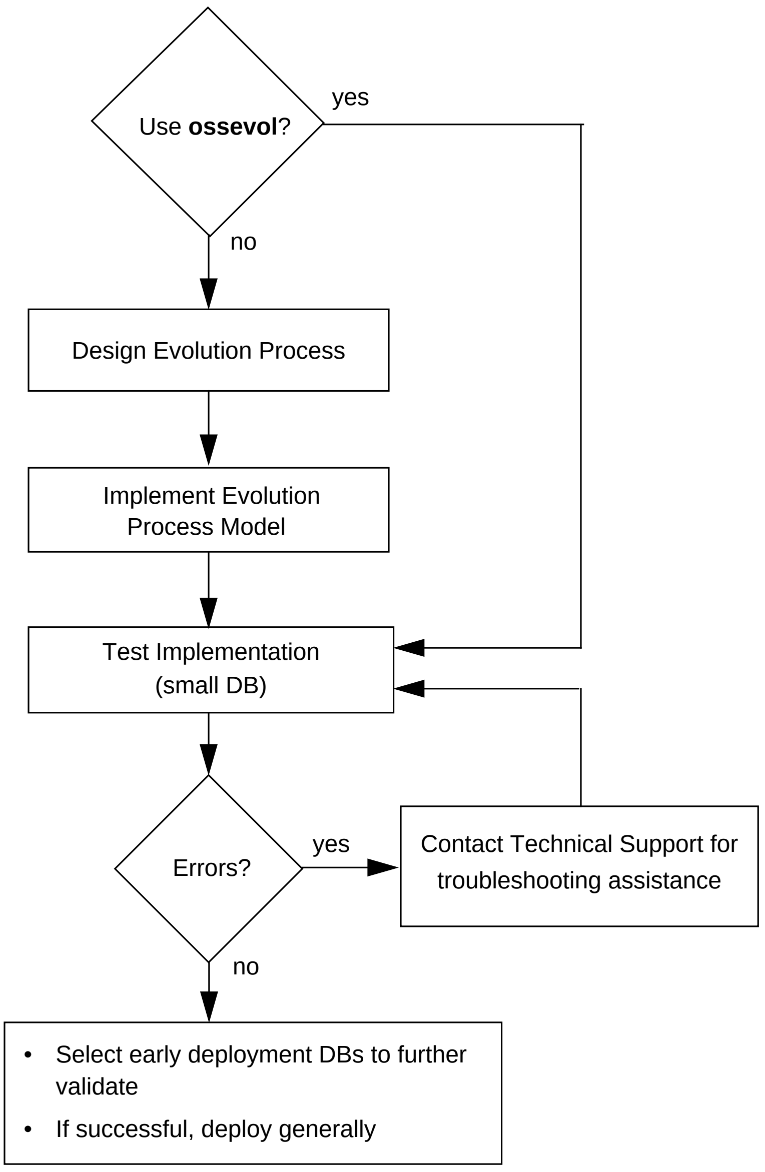
3 Implement your design.

4 Test your implementation and troubleshoot.

5 If the facility is to be used to upgrade databases currently in use, obtain some active databases for predeployment validation.

6 Limit your initial deployment to validated customers, followed by general deployment to all customers.

Schema evolution decision tree


Use **ossevol**?
yes
no
Design Evolution Process
Implement Evolution Process Model
Test Implementation (small DB)
Errors?
yes
Contact Technical Support for troubleshooting assistance
no
• Select early deployment DBs to further validate
• If successful, deploy generally

Schema Evolution with **ossevol**

The **ossevol** utility modifies a database and its schema so that it matches a revised application schema. It handles many common cases of schema evolution. Running the **ossevol** utility changes the physical structure of your database, so the importance of backing up your database before running this utility is critical.

Use this utility when you are performing simple operations such as adding or deleting data members that do not require a special evolving application.

Evolving schemas that contain dictionaries

To evolve the schema of a database that contains instances of **os_Dictionary** or **os_rDictionary**, you cannot use the **ossevol** utility; you must create a schema evolution application. See Implementing Schema Evolution on page 208.

ossevol Options

The **ossevol** options are described in the following table, and further information is available in *ObjectStore Management*, Chapter 4, Utilities. Object Design recommends that you use the **-task_list** option to make sure you understand the steps that the evolution will be taking before actually performing the schema evolution.

-task_list *filename*	Specifies that the **ossevol** utility should produce a task list and place it in the file specified by *filename*. Use **-** (hyphen) for **stdout**. When you specify this option, ObjectStore does not perform schema evolution. The task list consists of pseudofunction definitions that indicate how the migrated instances of each modified class would be initialized. This allows you to verify the results of a schema change before you migrate the data.
-classes_to_be_removed *class-name(s)*	Specifies the names of the classes to be removed.
-classes_to_be_recycled *class-name(s)*	Specifies the names of the classes whose storage space can be reused. By default, the storage associated with all classes is recycled.

-drop_obsolete_indexes { *yes* | *no* }	Specifies whether or not obsolete indexes encountered in the course of the evolution should be dropped. The default is **no**, which means that they are not dropped.
-local_references_are_db_relative	Specifies that all local references are relative to the database in which they are encountered. The default is **no**.
-resolve_ambiguous_void_pointers	Resolves ambiguous void pointers to the outermost enclosing collocated object. The default is **no**.
-upgrade_vector_headers	Upgrades the representation of vector objects in the evolved database to a format that allows them to be accessed by clients built by different types of compilers. You do not need to convert vector objects if the database will be accessed only by applications that were compiled with the same type of compiler. This option is for databases being used in an environment that includes multiple types of compilers. It is also useful if you are switching from **OSCC** (the ObjectStore C++ compiler) to a native compiler that uses vector headers, such as SGI C++. Use this option only with databases that meet at least one of these conditions: • Created before ObjectStore Release 5.1 • Built by applications compiled with a cfront or cfront-derived compiler You do not need to use this option if you only intend to access the schema from applications that were compiled with cfront, since cfront does not need vector headers.
-explanation_level *n*	A number from **1** to **3**; primarily an internal debugging aid.

Once you have completed the schema evolution process, you can use **osscheq** (see Using osscheq to Verify Schema Changes on page 206) and **osverifydb** (see Using osverifydb to Verify Pointers and References on page 206) to compare the old and new schemas, and to verify that the pointers and references are sound.

Using osscheq to Verify Schema Changes

The **osscheq** utility is useful in detecting whether a change to a schema causes it to be incompatible with the other schemas in an application. Check for any incompatibilities immediately following schema evolution. When schemas are not compatible, execution of the application fails because of a schema validation error.

Invoke the **osscheq** utility as follows:

osscheq *test1 test2*

Comparison technique

The comparison technique depends on the types of schemas being compared. When comparing compilation or application schemas, ObjectStore uses the technique used by the schema generator when building compilation or application schemas. When one of the schemas being compared is a database schema, the comparison technique is the same as that used to validate an application when it accesses a database.

This form of checking is the minimal checking required to ensure that the application and the database use the same layout for all shared classes.

A complete description of **osscheq** and other ObjectStore utilities can be found in Chapter 4, Utilities, of *ObjectStore Management*.

Using osverifydb to Verify Pointers and References

This utility verifies all pointers and references in a database. Using this utility prior to attempting schema evolution and again after schema evolution establishes that the database pointers and references are sound. The options available for this utility and additional information are available in *ObjectStore Management,* Chapter 4, Utilities.

Also run any application-specific verification tools available. This is particularly important because **osverifydb** can only detect problems at a fairly low semantic level.

Designing a Schema Evolution Application

The most important factor in planning schema evolution is the time and attention you apply to the details involved in your particular application. Plan the schema evolution of your application with the utmost care. What are the *specific* objectives your schema evolution application must accomplish?

Determine, for example, if you must write an evolution application or if you can accomplish your objective using the ObjectStore utility **ossevol**. For the most part, if you are adding or deleting a data member, you might make use of this utility. This depends on the kind of data member, however. A data member that requires a nondefault value (nonzero), or the need to add a C++ reference, would require that you write a special schema evolution application.

Example Using ossevol for Schema Evolution

Changing a class value

As a simple illustration, suppose that a customer database uses **address** objects that have a field for a five-digit ZIP code. With the schema evolution facility, you can change the definition of the class **address** so that the value type of the data member **address::zip_code** is **char[9]** instead of **char[5]** — or, better yet, you can use a new or existing class, **string** or **zip_code**, to serve as the new value type in place of **char[5]**.

If you invoke schema evolution on the database, using the new class definitions, **ossevol** modifies the database's schema and changes each **address** object to use the new type of object for its **address::zip_code** (adjusting the size of the **address::zip_code** field, if necessary).

The value contained in the new field can either be determined by a user-defined routine that has access to the unevolved data, or it can be set automatically. When set automatically, the new value is set by assignment from the old value, if possible, and otherwise it is set to a default value (such as 0 or the result of executing a constructor that initializes each member to 0).

Implementing Schema Evolution

If you have determined that you cannot achieve your objectives using the ObjectStore utility **ossevol**, you must then plan the specific steps required for your application's schema evolution. Be sure to include all the phases of preparation and testing described in this section.

Because of the potential complexity of the schema evolution process, it is important to incorporate as many safeguards as possible into your schema evolution application, to test it thoroughly using small databases, and to validate that the schema evolution has been successful. This chapter provides basic guidelines, examples, and validation techniques.

For explanations and examples of more complex schema evolution issues, see Chapter 9, Advanced Schema Evolution, in the *ObjectStore Advanced C++ API User Guide*.

The Schema Evolving Application

What to include

Regardless of the evolution you intend to perform, make sure you plan to include the following in your evolution application:

- Before evolution starts, use **osverifydb -all** to make sure the database returns a **0** result code indicating no errors (that is, you are starting with a clean, error-free database).
- Tag any databases with a state block in which the states are, for example, operational, evolving, or validating.
- Tag any databases that have version information that is only updated after an evolution has been deemed successful (**osverifydb -all** returned **0**).
- In your application, include an instantiation for each **os_Dictionary** or **os_rDictionary** instantiation in the database being evolved.
- Ensure that an application tests the version information and the state information before resuming normal operations.

What to avoid

Delete **os_cursor** objects before schema evolution (schema evolution cannot handle **os_cursor** objects).

Unions require a very complicated custom schema evolution application.

Validation Activities

The following is a checklist of validation tasks you should perform to confirm that the schema evolution actually accomplished your objectives and did not make unexpected alterations to the database.

- The first part of the validation stage should rerun **osverifydb -all** and again return a **0** result code indicating no errors.
- Inspect the database in light of the the semantics of the data stored there (as much data as possible should be validated).
- If the database is very large, do some statistical probing of the data.

Testing

Some additional testing you can perform to ensure that the database is as you expect includes

- Write a test harness to exercise the database completely.

Troubleshooting

If you encounter difficulties when performing or testing the results of any schema evolution operation, you must debug carefully with the assistance of Object Design Technical Support. You must supply support with the

- Preevolution database
- Your schema evolution application
- Stack trace of the time of failure

Deploying Schema Evolution

Once you are satisfied that the schema evolution application you have designed and tested is accomplishing your objectives successfully, you can begin deployment. It is advisable to do this in phases as well.

First get some of your customers' databases and initiate schema evolution on those databases. Validate the results.

General deployment tip

Be sure to deploy in stages by validating that your schema evolution application works on several small customer databases before you make it generally available.

Chapter 9
Using Asian Language String Encodings

There are many standards for encoding Asian characters. In Japan, for example, five encodings are in broad use: JIS, SJIS, EUC, Unicode, and UTF-8.

Usually an application uses one encoding for all strings to be stored inside a database. The encoding chosen is most often the one used in the operating system of the ObjectStore client.

However, if the application has heterogeneous clients using a variety of encodings, conversion from one encoding to another is necessary at some point. The clients could be traditional ObjectStore client processes or thin-client browsers that emit data in different encodings.

The Class Library: os_str_conv

This class library provides conversion facilities for various Japanese language text encoding methods: EUC, JIS, SJIS, Unicode, and UTF8.

The library provides a facility 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 problem for Internet applications.

The most common application of this class is conversion between EUC and SJIS to provide sharing of data from UNIX <-> Windows applications. JIS is commonly used for email. Applications normally store data in a homogeneous format inside a database, and incoming strings are converted as required before they are persistently allocated. Outgoing strings can also be converted to the client’s native encoding. For Web applications this outgoing conversion is usually not necessary since internationally aware browsers (Netscape 2.0 and above, for example) can automatically detect and convert various incoming formats themselves.

The class library currently consists of a single class, **os_str_conv**, instantiated once for each conversion path required:

```
os_str_conv(encode_type dst, encode_type src=automatic);
```

Where

```
enum encode_type {    /* string encode type ------------------- */
UNKNOWN=0,            /* convert or automatic detect fail        */
AUTOMATIC,            /* detect automatically                    */
AUTOMATIC_ALLOW_KANA,
                      /* detect automatically, allow half-width-kana */
ASCII,                /* ASCII                         */
SJIS,                 /* Shift-JIS                     */
EUC,                  /* EUC                           */
UNICODE,              /* Unicode (can’t automatic detect)   */
JIS,                  /* JIS                                */
UTF8                  /* UTF-8 (can’t automatic detect)     */
/* add new encode type here ! */
};
```

Here is an example. Given an instance of **os_str_conv**, such as

```
os_str_conv *sjis_to_euc = new os_str_conv(os_str_conv::EUC,
os_str_conv::SJIS);
```

A conversion can be done on **char* sjis_src**:

```
char *euc_dest = new char[sjis_to_euc->get_converted_size(
    sjis_src)];
```

```
sjis_to_euc->convert(euc_dest, sjis_src);
```

The call to **get_converted_size()** is not strictly required; it is provided for the convenience of the user to allocate buffers of appropriate size. Because it requires examination of the entire source string, time to complete it is proportional to the source string length.

Automatic Detection of a Source String Encoding

Sometimes, it is not possible for an application to know the encoding of a given source string. **os_str_conv** provides methods that can analyze a given string and determine its encoding. For example:

```
os_str_conv *to_euc = new os_str_conv(
    os_str_conv::AUTOMATIC); len =
         to_euc->get_converted_size(unknown_src);
    if (len)
    {
    char *euc_dest =
        new char[to_euc->get_converted_size(unknown_src)];
    to_euc->convert(euc_dest, sjis_src);
     }
    else
    {
// couldn't convert -- application needs to handle this!
    }
```

Important Note: The autodetector is not guaranteed to work in all cases.

If it fails inside **get_converted_size**, **get_converted_size** returns **0** to indicate the failure. Be careful not to allocate strings based on its return value without checking for failure!

Unfortunately, no automatic detection algorithm can correctly distinguish EUC from SJIS in all cases because of overlap in their assignment ranges. Clever algorithms exploit patterns typical of real text. This implementation is reasonably straightforward. The most difficult problem (distinguishing between SJIS half-width kana and EUC) is avoided by asking the user to choose between the two possible interpretations. In nearly all cases, **os_str_ conv::AUTOMATIC** is the appropriate setting.

In practice, the problems of ambiguity are not likely to affect applications, since usually incoming text is all in the single encoding defined by the operating system used when generating it. Autodetect can be used at the beginning of a session only, and it can be reasonable to assume that it will not change.

As mentioned earlier, there are areas in the EUC and SJIS encodings that overlap, and so a given string might be valid in either encoding. This makes autodetection ambiguous.

There are two ambiguous cases:

- The half-width kana of SJIS. It is possible for a string consisting entirely of bytes in this range to be either SJIS or EUC. This is the most troublesome case.

- An obscure range of SJIS and EUC that overlaps. The characters represented by this range are rarely used, so it is highly unlikely that a string would consist entirely of such characters.

Detection is handled according to these rules:

1 The algorithm examines each character of the string in sequence until the encoding is determined. Therefore, a string beginning with an unambiguous substring followed by an ambiguous substring is detected according to the first substring.

2 Strings consisting entirely of the second ambiguous type are handled as unknown. As mentioned, this case is very unlikely.

3 All SJIS half-width kana are single-byte encodings. Therefore, a string consisting entirely of an odd number of bytes in the SJIS half-width kana range is considered SJIS.

4 A string beginning with an even number of bytes in the SJIS half-width kana range is ambiguous until the following characters are examined according to normal detection rules.

5 A string beginning with an odd number of bytes in the SJIS half-width kana range requires special examination of the last character. If this is an EUC first-byte code, and it is followed by a valid EUC second-byte code, then the string is EUC. However, if the following code is not a valid EUC second-byte (it might be ordinary ASCII), then the final character is interpreted as SJIS half-width kana and the string is interpreted as SJIS.

6 A string consisting entirely of an even number of bytes in the SJIS half-width kana range is ambiguous. It is quite possible for such a string to appear in real applications. The **os_str_ conv::automatic** setting causes the autodetector to interpret this case as SJIS. However, if **os_str_conv::automatic_allow_kana**, this case is interpreted as unknown. Ojbect Design believes that the SJIS interpretation is correct for most cases.

Japanese developers are aware of the problems handling the half-width SJIS kana, and so they try to avoid them by using full-width SJIS kana instead.

Unlike EUC and SJIS, JIS is a modal encoding that uses <Esc> to enter and exit from multibyte mode. Detecting JIS strings is accomplished by searching for these <Esc> characters.

How to Instantiate the Converter

The class **os_conv_str** must be instantiated once for each conversion path required for your application.

Guidelines for Extensions to os_str_conv

Users can extend this class by inheriting from it. This could be useful for developers who want to override the existing autodetector.

Additional encodings can be appended to the existing enumeration. Note that **os_str_conv** depends on the ordering of the existing encodings, so if you extend **os_str_conv**, additional encodings must appear after the ones already provided.

What Are the Different Modes and Their Meanings?

Notes on encodings

For most purposes, there is a one-to-one mapping for characters to and from each of these encodings, so no semantic information is lost during conversion. There are four exceptions to this rule:

- EUC and Unicode are a superset of SJIS and so roundtrip EUC / Unicode<->SJIS is not possible for all EUC / Unicode Japanese characters.
- There are a handful of cases of pairs of SJIS characters that map to a single character in Unicode.

  The second class of exceptions is considered extremely minor in practice, and is the result of different editions (1983 and 1990) of the JIS as the basis of SJIS and Unicode.
- Third, SJIS contains some special characters that are printable on Windows. Although mappings are defined for EUC, attempts to view them on X-windows, at least, fail because the fonts in use do not provide glyphs for those codes. There are no encodings for these characters in Unicode.
- Lastly, JIS defines multiple ways to express a character (the base semantic unit), so a conversion from JIS to another encoding and back to JIS is not guaranteed to return an identical binary string. However, the meaning of the string (in the sense of the way it would appear if printed on a screen) is the same.

Variations Among Standard Character Mappings

The Unicode Consortium has published a general mapping from Shift-JIS to Unicode. However, actual implementations of the standard mapping differ slightly by platform and vendor. The **os_str_conv** class is implemented with a default mapping according to the Unicode Consortium standard, and also provides a means by which any mapping entry can be overridden at run time by a client application.

The deviations in mapping tend to be quite small. For example, here is a table that shows the incompatibility of the Unicode Consortium standard and the maps that Microsoft uses in Windows NT:

SJIS Code	*Unicode Consortium Mapping*	*Microsoft Mapping*
\ 5C	00A5 YEN SIGN	005C REVERSE SOLIDUS(*)
~ 7E	203E OVERLINE	007E TILDE
^[$B!@(B 81,5F	005C Reverse solidus	FF3C FULLWIDTH REVERSE SOLIDUS
^[$B!A(B 81,60	301C WAVE DASH	FF5E FULLWIDTH TILDE
^[$B!B(B 81,61	2016 DOUBLE VERTICAL LINE	2225 PARALLEL TO
^[$B!](B 81,7C	2212 MINUS SIGN	FF0D FULLWIDTH HYPHEN-MINUS
^[$B!q(B 81,91	00A2 CENT SIGN	FFE0 FULLWIDTH CENT SIGN
^[$B!r(B 81,92	00A3 POUND SIGN	FFE1 FULLWIDTH POUND SIGN
^[$B"L^(B 81,CA	00AC NOT SIGN	FFE2 FULLWIDTH NOT SIGN

Instructions on Overriding Particular Mappings

How to modify standard encodings

To allow an application to modify the standard encoding on the fly, there is the following interface:

```
class os_str_conv {
public:
...
        struct mapping {
            os_unsigned_int32 dest;      /* destination code */
            os_unsigned_int32 src;        /* source code      */
         };
        int change_mapping(mapping table[],size_t table_sz);
    ...
};
```

You can modify an existing instance of **os_str_conv** (whether heap- or stack-allocated) by calling **os_str_conv::change_ mapping()**. Actually, internal mapping tables, shared by all instances of **os_str_conv**, are never modified. The additional mapping table information is stored to provide override information 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 return **-1** from **change_mapping()** to indicate this error condition.

The **change_mapping()** method takes the following two parameters:

- **os_str_conv::mapping_table[]**

  This is 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 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**

  This is the number of elements in the **mapping_table**. The user should take care that this is not the number of bytes in the array.

Example

Here is an example of a Microsoft SJIS->Unicode mapping.

```
os_str_conv::mapping mapping[] = {
    {0x0000005C,0x0000005C},
    {0x0000007E,0x0000007E},
    {0x0000815F,0x0000FF3C},
    {0x00008160,0x0000FF5E},
    {0x00008161,0x00002225},
    {0x0000817C,0x0000FF0D},
    {0x00008191,0x0000FFE0},
    {0x00008192,0x0000FFE1},
    {0x000081CA,0x0000FFE2},
};

void func(char* input,char* output) {
        ...
os_str_conv sjis_uni(os_str_conv::SJIS,os_str_conv::UNICODE);
    sjis_uni.change_mapping(mapping,sizeof(
        mapping)/sizeof(mapping[0]));
    sjis_uni.convert(output,input);
        ...
}
```

In this example, **mapping[]** is a global, but a stack allocation would work as well.

Byte Order

Since Unicode is a 16-bit quantity, byte order depends on platform 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 this:

```
encode_type convert(char* dest, const char* src);

encode_type convert(os_unsigned_int16* dest, const char* src);

encode_type convert(char* dest, const os_unsigned_int16*
```

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.

Overhead

Using overrides to the string conversion function incurs the following overhead:

- Some memory is consumed by the sorted override map.
- Some time is consumed when sorting the map at **change_mapping** time.

- Some time is consumed when converting characters because of the additional lookup.

Restrictions

Not all conversion combinations are possible. For example, it is impossible to convert Unicode to ASCII. This implementation guards against nonsensical requests, but developers who extend it should take care for such cases. Of course, Japanese to ASCII conversion is only possible on the ASCII subset of characters in the Japanese encodings. Attempts to convert Japanese strings to ASCII result in the return of an error condition.

Autodetect only detects SJIS, JIS, and EUC. Do not feed the autodetector Unicode or UTF-8 strings.

The EUC<->Unicode converter only works for characters in the SJIS set. While this might sound perverse, it is reasonable for actual applications, since characters outside the SJIS set are extremely rare.

Users should be aware that the 0 to 127 range of single-byte SJIS characters is *not* ASCII, even though the characters look like ASCII. This range is known as *JIS-Roman*. Specifically, the characters {'\', '~' , '|'} have different meanings. The practical significance is that the map of characters [0 to 127] from ASCII->Unicode->SJIS is not an identity.

Performance Notes

EUC and SJIS are very closely related since they both are based on the JIS ordering. Therefore, conversion between these requires no table lookup.

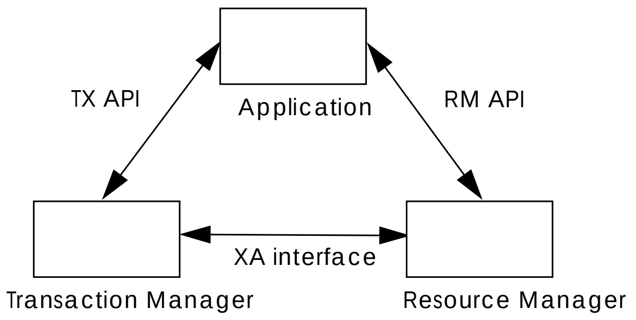
JIS conversion requires simple parsing for <Esc> characters. Once stripped of <Esc> characters, you can convert the multibyte sequences to EUC by setting the highest bit.

Chapter 10 Support for the XA Standard for Transaction Processing

ObjectStore supports X/Open's transaction demarcation protocol (known as XA).

Distributed Transaction Processing model

XA is a set of services that is part of the X/Open Distributed Transaction Processing (DTP) model. The model consists of three components: an application, a transaction manager, and a resource manager.



Transaction manager

In the DTP model, transaction demarcation is controlled by a *transaction manager* (TM).

The transaction manager can coordinate distributed transactions in multiple database systems so that, for example, one transaction can involve one or more processes and update one or more databases. The databases could be multiple ObjectStore databases or they could be incompatible databases, such as ObjectStore and Sybase.

Two-phase commit

Transactions under the control of a transaction manager are made through a two-phase commit process. In the first phase, transactions are prepared to commit; in the second, the transaction is either fully committed or rolled back.

Resource manager

An ObjectStore client acts as a resource manager (RM). Resource managers must be registered with the transaction manager.

Interface for applications

Applications are written using the transaction manager’s API. XA is the interface between the transaction manager and the resource manager and is not visible to application programmers.

Transactions in the DTP Model

A *global transaction* is a metatransaction managed by a transaction manager and possibly involving multiple resource managers. Such transactions have globally unique identifiers generated by the transaction manager. Note that DTP global transactions are distinct from ObjectStore global transactions.

A *transaction branch* is a transaction from a resource manager's point of view.

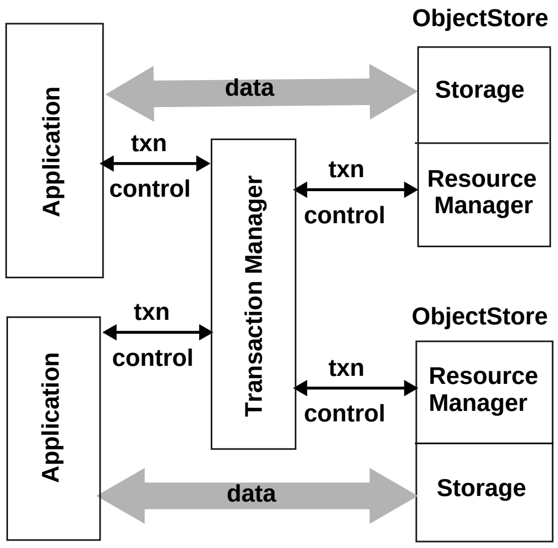
A global transaction can consist of many transaction branches in a mix of resource managers. Multiple branches belonging to the same global transaction can exist on a single resource manager.

XA transactions are identified by XIDs (universally unique identifiers for global transactions). A branch has a globally unique ID composed of the global transaction ID and a branch ID.

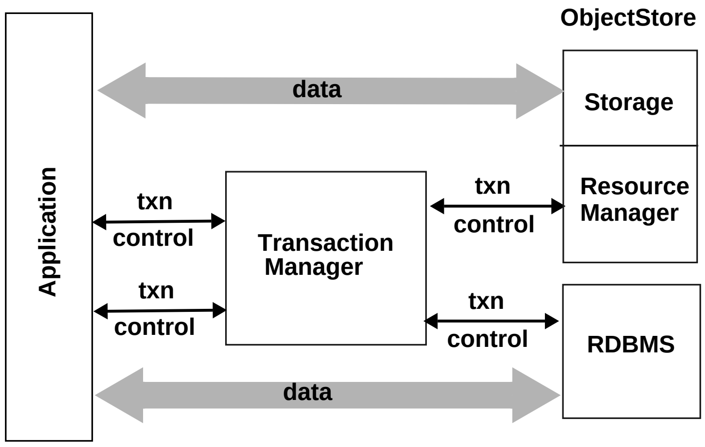
XA transactions are always of the type update.

XA is also integrated into the OMG Object Transaction Service (OTS), which is based on DTP. The examples given show the use of Iona’s OrbixOTS implementation of OTS.

An application that uses a transaction manager must use the TM's interface, for example, the OMG Object Transaction Service (OTS), to begin and end transactions. Other database operations can be done using the database systems' proprietary APIs. When recovery is required, the TM manages the process, using the XA interfaces to roll back or commit transactions that were in progress.



ObjectStore clients



ObjectStore and RDBMS database

Registering ObjectStore as a Resource Manager

You must register ObjectStore as a resource manager with your transaction manager.

Registering a resource manager provides basic identification information used in initialization and in processing transactions.

The information you must provide to register ObjectStore as a resource manager is packaged in a global data structure of type **xa_switch_t**. This data structure is defined in **libos** with the name **ObjectStore_xa_switch**.

In addition, for each ObjectStore resource manager, you must identify the hosts that will run ObjectStore Servers for applications that use the transaction manager.

There are several ways to register a resource manager. See your individual vendor documentation for specific information.

The **Encina::Server::RegisterResource** function

The next example uses the OrbixOTS as the transaction manager and describes how to register a resource manager by passing a set of arguments to the **Encina::Server::RegisterResource** function.

Information for the following parameters is required::

xaSwitchP	A pointer to a data structure of type **xa_switch_t**. Specify the symbol **ObjectStore_xa_switch** for ObjectStore resource managers.
openString	A string specific to the resource manager. For ObjectStore, this is a space-separated list of hosts running ObjectStore Servers that might be contacted by the ObjectStore client that is the resource manager. This list of Servers is queried during recovery operations. An attempt to contact a Server not in the list is considered an error.
closeInfo	Empty string.
isThreadAware	Specifies the type of thread support. For ObjectStore resource managers, use the value False.

Intialization Example

The following example illustrates sample initialization code.

```
extern struct xa_switch_t ObjectStore_xa_switch;
//  XA_OSTORE_SERVERS should be a space-delimited
//  list of ObjectStore Server host names. For example,
// "host.domain.com host2.domain.com ..."

char *openStringP = getenv("XA_OSTORE_SERVERS");

Encina::Server server;

// Register the database as a resource manager

server.RegisterResource(
    &ObjectStore_xa_switch, openStringP, "");
```

Using the Transaction Manager

Once you have registered ObjectStore as a resource manager, you can use the transaction manager's interface to start and commit transactions.

Nested transactions

All transactions controlled by the transaction manager must be the top-level transaction. Nested DTP transactions are not allowed. Nested ObjectStore transactions are allowed within an XA transaction. These can be started and committed using the native ObjectStore interfaces.

If your program tries to start a transaction through the transaction manager when another transaction is already in progress, the transaction manager receives an error.

If your programs attempt to use the regular ObjectStore interface to commit or abort a transaction that was started by a transaction manager, one of the exceptions err_commit_xa or err_abort_xa is generated.

Concurrency modes

When ObjectStore is in use, the concurrency mode must be **SERIALIZE_TRPCS_AND_TRANSACTIONS**. ObjectStore can run only one transaction at a time, and this concurrency mode prevents the transaction manager from trying to start multiple concurrent transactions. Attempts to use other concurrency modes can result in the transaction manager’s getting a deadlock error at the XA interface when starting a transaction.

Here is an example of using **SERIALIZE_TRPCS_AND_ TRANSACTIONS**:

```
server.Listen(Encina::Server::SERIALIZE_TRPCS_AND_TRANSACTIONS);
```

Two-Phase Commit and Recovery

The ObjectStore client refers to XA transactions by XID. The XID is saved in the Server's transaction log so that the Server, when queried after a crash, can provide a list of the XIDs of transactions that were *prepared* (phase 1 commit) but not *committed* (phase 2 commit).

Recovery is through the transaction manager and occurs during the initialization process for a DTP application.

Restrictions

There are some restrictions on the use of ObjectStore’s native transaction interfaces when used with distributed transaction processing. Other ObjectStore interfaces can be used as usual.

- DTP global transactions can have multiple transaction branches in separate processes that access the same resource manager. ObjectStore supports loosely coupled global transactions, in which the different transaction branches of a global transaction are regarded as distinct transactions, and are therefore subject to deadlock with one another if they attempt to access the same data.
- Multi-Server backup does not synchronize with XA transactions. With solely ObjectStore transactions, a transaction-consistent backup of multiple Servers can be taken even while two-phase transactions are in progress. This is not true for XA transactions.
- You cannot use checkpoint / refresh (**os_transaction::checkpoint()**) within a DTP transaction

because top-level transactions must be handled using transaction manager calls.

Chapter 11
Component Schemas

Component schemas allow you to incorporate one or more DLLs in an ObjectStore application.

A component schema is a schema describing class types for a particular DLL. A DLL schema is to a DLL as an application schema is to an ObjectStore application.

Like a DLL, a DLL schema can be loaded and unloaded dynamically at run time.

This chapter introduces two new terms: *program schema* and *complete program schema*.

Component Schema and Application Schema

A component schema, also referred to here as a DLL schema, is a schema that is contained within a DLL. It plays the same role for the DLL as an application schema plays for an application. Like a DLL, a DLL schema can be loaded and unloaded dynamically at run time. Unlike an application schema, multiple DLL schemas can be in effect at the same time in a single program.

Differences between an application schema and a component schema

A component (DLL) schema is a type of application schema with some additional properties. The file name extension **.adb** is used for both application schemas and DLL schemas. DLL schemas are generated by **ossg** just as application schemas are.

A program schema is an application or DLL schema. A complete program schema is all the loaded program schemas for an application.

When a type name is defined by more than one program schema, all definitions of the type must be the same.

Component schemas differ from application schemas in these ways:

- A DLL schema can be loaded and unloaded dynamically at run time.
- Multiple DLL schemas can be in effect at the same time in a single program.
- DLL schema can be installed only in incremental mode.

Generation of DLL schemas is fundamentally the same as for application schemas. See Generating an Application or Component Schema in *ObjectStore Building C++ Interface Applications* for further information.

All ObjectStore utilities that can be used on application schemas can be used for component schemas.

Uses for Component Schemas

Some typical uses of DLL schemas are as follows:

- A class library or object manager for a particular kind of persistent objects is often packaged as a DLL. Packaging the associated schema (the classes implemented by the library and all classes reachable from them) as a DLL schema allows the DLL / DLL schema pair to serve as a stand-alone unit. Although an application statically linked to the DLL will have the externally visible classes of the DLL in its application schema, there might be additional internal classes that are only in the DLL schema. Also, applications like the ObjectStore Inspector, that provide generic database operations without having specific persistent classes built into them, can automatically load the DLL and its schema when operating on a database that contains persistent instances of the classes implemented by the DLL.
- A program can switch on the fly between two different implementations of a persistent class by unloading one DLL that implements the class and loading a different DLL that implements a class with the same name. The corresponding changes to the program schema happen by loading and unloading DLL schemas. For this to work, the class being changed must not be in the application schema and the two implementations of the class must have identical run-time layouts.
- A component-based system, such as a web server that loads plug-ins or extensions, can load ObjectStore-based components packaged as DLLs. In this case, the application is completely unaware of ObjectStore and there is no application schema. The DLL schema of the component serves the traditional role of an application schema. This scenario is more likely than the first two to involve unloading of DLLs and DLL schemas when components are no longer active.

How to Use Component Schemas

An example of four ways to use component schemas is given in the ObjectStore examples directory. The example can be found in these locations:

- On Windows platforms: **\packages\examples\flights**
- On Solaris platforms: **$OS_ROOTDIR/examples/flights**

The example illustrates four ways to use DLLs with ObjectStore.

The first example, **flights**, uses a standard application schema.

The second example, **flights2**, uses a component schema generated for the DLL **flight_cs** with the schema source file macro **OS_SCHEMA_DLL_ID( )**. The component schema is loaded and unloaded automatically.

The third example, **flights3**, also uses a component schema in the same way as the previous example and additionally illustrates how a DLL can be dynamically loaded with an application. This example uses automatic load reporting, as shown in the following excerpt:

```
    ...
    const char *flight_cs_id = "DLL:flight_cs";  // DLL Identifer
...

...
    const char *flight_cs_symname = "flight_db_component";
    os_DLL_handle dll_handle = os_null_DLL_handle;
    os_boolean caught_except = false;

    // Load the flight_cs component.
    TIX_HANDLE(err_DLL_not_loaded) {
        // This is equivalent to calling ‘dlopen’ on Solaris2 or
        // ‘LoadLibrary’ on WIN32 platforms.  objectstore::load_DLL
        // is provided as a convenience for application developers.
    dll_handle = objectstore::load_DLL(flight_cs_id, true);
    }
    TIX_EXCEPTION {
        caught_except = true;
    }
    TIX_END_HANDLE
    if (caught_except || (dll_handle == os_null_DLL_handle)) {
        cout << "Error: " << tix_handler::get_report() << ‘\n’;
        cout << "Cannot load component: " << flight_cs_id << ‘\n’;
        return 2;
    }

    // Look up the component's entry point
    caught_except = false;
    TIX_HANDLE(err_DLL_symbol_not_found) {
        // Same as calling 'dlsym' on Solaris2 or 'GetProcAddress' on
        // WIN32 platforms.  Provided as a convenience for application
        // developers.
    cfp = (int(*)(const char*))
    objectstore::lookup_DLL_symbol(dll_handle,
                            flight_cs_symname);
    }
    TIX_EXCEPTION {
        caught_except = true;
    }
    TIX_END_HANDLE
    if (caught_except || !cfp) {
        cout << "Error: " << tix_handler::get_report() << '\n';
        cout << Cannot locate symbol: " << flight_cs_symname <<'\n';
    return 3;
     }
...
```

The fourth example, **flights4**, has the features of the previous example, but does not use automatic loading of the component schema. Instead it shows how to manually control how component schemas are loaded and unloaded. In the schema source file you see

```
// Component schemas need a DLL identifier so we define
// one using this schema macro.
OS_SCHEMA_DLL_ID("DLL:flight2")

// Shut off automatic load/unload reporting.
OS_REPORT_DLL_LOAD_AND_UNLOAD(false)

// Tell ossg to use this variable name for the schema info.
OS_SCHEMA_INFO_NAME(flight_cs2_dll_schema_info)
```

main.cpp file

And in the **main.cpp** file:

```
...
    const char *flight_cs_id = "DLL:flight_cs2";  // DLL Identifier
    os_DLL_schema_info *flight_cs2_sch_inf = NULL;
    os_schema_handle *cs2_sh;
...

...
    // Via a macro in the schema source file, we arranged for the
    // schema generator to place a symbol name that we specified to
    // be the name of the schema info structure
    //(os_DLL_schema_info *).
    // We simply look up the symbol in the DLL to get at it.
    caught_except = false;
    TIX_HANDLE(err_DLL_symbol_not_found) {
        flight_cs2_sch_inf = (os_DLL_schema_info *)
        objectstore::lookup_DLL_symbol(dll_handle,
                    "flight_cs2_dll_schema_info");
    }
    TIX_EXCEPTION {
        caught_except = true;
    }
    TIX_END_HANDLE
    if (caught_except || !flight_cs2_sch_inf) {
        cout << "Error: " << tix_handler::get_report() << '\n';
        cout <<"Can't locate symbol: flight_cs2_dll_schema_info\n";
    return 4;
    }

    // Automatic load reporting has been disabled for this component
    // schema(flight_cs2).  Which means that the schema for this DLL
    // does not automatically load when put in use.  So we must
    // manually set up the loading process here.

    // Tell ObjectStore that this DLL has been loaded.
    cs2_sh = &flight_cs2_sch_inf->DLL_loaded();
...
```

Unload the DLL

After using the component, explicitly unload the DLL:

```
...
    //
    // Unload the component
    //
    // Tell ObjectStore that this DLL is about to be unloaded.
    flight_cs2_sch_inf->DLL_unloaded();

    objectstore::unload_DLL(dll_handle);
...
```

Building Component Schemas

Solaris

On Solaris platforms, to build an application with static linking, use a command of the following form:

```
CC foo.cpp foolib.cpp -ldl -o foo
```

To build an application with dynamic linking, use a command of the following form:

```
CC -G foolib.cpp -o foolib.so
CC foo.cpp foolib.so -ldl -o foo
```

To build an application with run-time library loading, use a command of the following form:

```
CC -G foolib.cpp -o foolib.so
CC foo.cpp -ldl -o foo
```

To build with dynamic linking and an explicit call to the library,

```
CC -G foolib.cpp -o foolib.so
CC foo.cpp foolib.so -ldl -o foo
```

DLL Loading and Unloading

Typically DLLs are loaded and unloaded by means of operating system calls. ObjectStore provides two additional methods to load and unload DLLs into ObjectStore applications.

A platform-independent interface:

- **objectstore::load_DLL()**
- **objectstore::lookup_DLL_symbol()**
- **objectstore::unload_DLL()**

An interface that enables ObjectStore to automatically load DLLs when a database is put into use:

- **objectstore::get_autoload_DLLs_function()**
- **objectstore::set_autoload_DLLs_function()**

These check whether the DLL is loaded or queued to load. If not, they call **objectstore::load_DLL()**, which checks each DLL ID and loads it.

See the *ObjectStore C++ API Reference* for more information on these interfaces.

DLL Load and Unload Reporting

When a DLL that has a DLL schema is loaded or unloaded, it must report that fact to ObjectStore so ObjectStore can load or unload the DLL schema. You do this reporting by calling the functions described below. In general, the actual schema loading or unloading is deferred until a later time.

In the simplest case, calls to these functions are automatically generated by **ossg** and inserted into the DLL as initialization and termination functions. The application developer does not have to do anything to implement this reporting. If you want to explicitly control this, specify in the schema source file that the automatic calls should be disabled and then write code that makes the calls. For example, you could set some ObjectStore parameters or the pathname of the DLL schema database before calling **DLL_ loaded()**. This code could be in DLL initialization and termination functions, or could be in entry points to the DLL that are called according to the developer's specific protocol. See the *ObjectStore C++ API Reference* for further information on these functions:

- **os_DLL_schema_info::DLL_loaded()**
- **os_DLL_schema_info::DLL_unloaded()**
- **os_DLL_schema_info::add_DLL_identifier()**

DLL Identifiers

The DLLs that use component schemas must be assigned a *DLL identifier*. A DLL identifier is a generic way of identifying a DLL or catalog of DLLs. This identifier can be recognized by all platforms sharing a database that depends on the DLL.

A DLL identifier is a string of the form **prefix:suffix** where the prefix is a string that identifies the catalog of DLLs to be used and the suffix is something meaningful to that catalog. A colon can also appear in the suffix, but the first colon in an identifier is always interpreted as a separator.

ObjectStore provides the following built-in DLL identifier prefixes:

- **DLL:** followed by a platform-independent file name. It gets converted into a platform-specific file name, when it is assigned a suffix such as **.dll** or **.so**. These names are case sensitive. The operating system's usual library search stays in effect.

  On Windows, the conversion appends **.dll**. On Solaris, the conversion appends **.so**.

- **file:** followed by a platform-specific file name. It can be an absolute or relative path name, or just a file name subject to the usual search rules, on typical platforms. The case sensitivity of these names is determined by the platform's file system.

You can also create other DLL identifier prefixes if needed.

See the following for more information on uses of DLL identifiers:

- **os_database::get_required_DLL_identifiers()**
- **os_database::insert_required_DLL_identifier()**
- **os_database::insert_required_DLL_identifiers()**
- **os_database::remove_required_DLL_identifier()**

Schema Generation 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. To do this **ossg** generates code that calls **os_DLL_schema_info::DLL_loaded()** and **os_DLL_schema_info::DLL_unloaded()** from the DLL's initialization and termination functions.

OS_SCHEMA_DLL_ID

OS_SCHEMA_DLL_ID(*string*)

Component schema source file macro

Optional. 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

Required for component schema. 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**.

Creating a DLL Identifier Prefix

In some circumstances you might need to create a DLL identifier using a prefix other than the ObjectStore-supplied prefixes **DLL:**

and **file:**. Some examples of DLL catalogs for which there could be prefixes are

- DLL file
- DLL file (secure search of system paths only...)
- Windows ProgID (symbolic name, mapped in registry)
- Symbolic JavaBean ID, and so on

To inform ObjectStore how to understand a DLL prefix, create an instance of a subclass of **os_DLL_finder** and call its **register_** function with the prefix string. Be sure to unregister the prefix before deleting the instance. It is customary for each subclass of **os_DLL_finder** to know the prefix that it implements and have a constructor and destructor that call **register_** and **unregister** respectively.

You cannot register a DLL finder from a static constructor that could be called before **objectstore::initialize()** has been called. You must call **objectstore::initialize()** before doing anything with DLL finders, even registering them.

The argument to the **get()** function is a DLL identifier, not a prefix. The function finds the prefix by searching for a colon.

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.

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.

To compare two DLL identifier strings, call the static function **os_DLL_finder::equal_DLL_identifiers()**. It takes care of getting the prefixes, looking up the finder, and calling **equal_DLL_identifiers_same_prefix**. Looking up objects by DLL identifier calls **os_DLL_finder::equal_DLL_identifiers()** to compare identifiers that have the same prefix.

The **objectstore::load_DLL()** functions call **os_DLL_finder::load_DLL()** for the finder that implements the prefix of the specified **DLL_identifier**.

See **os_DLL_finder** in the *ObjectStore C++ API Reference* for further information.

Compiler Dope Damage

Compiler dope is additional information added to the run-time layout of an object by the compiler, beyond the nonstatic data members of the object. Compilers use compiler dope for several purposes, most notably to point to a table of virtual function implementations for the object's class. When ObjectStore brings a persistent object into memory from the database, it ensures that the object contains the correct compiler dope for the current program, for the compiler with which the program was compiled. 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.

Transient dope is the portion of the compiler dope that contains pointers to transient (nonpersistent) memory (that is, pointers to virtual function tables), and thus must be regenerated each time a persistent object is brought into the memory of a new program instance.

Dope damage is said to occur when the compiler dope of a cached persistent object becomes outdated as a result of loading or unloading DLL schemas. ObjectStore automatically detects dope damage when it occurs. You can select whether the response to dope damage when loading a DLL schema is to throw an exception or to repair the damage by regenerating compiler dope in all cached user data pages of affected databases. Note that this operation can take some time, so enable repair only when necessary. Dope damage when unloading a DLL schema occurs if there were any objects with virtual functions implemented in the unloaded DLL, so ObjectStore always repairs this form of damage.

Compiler dope can be damaged during the loading of a DLL schema in the following cases:

- When a persistent object is brought into virtual memory before the implementation of its class is loaded
- When a class implementation is redefined by loading a DLL at a point in time that the class is in actual use

You can choose whether ObjectStore throws an err_transient_dope_damaged exception or automatically repairs the damage when dope damage occurs. Repairing the damage incurs overhead. To avoid this, enable the exception for programs that do not expect dope damage to occur. If you expect programs that load and unload DLLs dynamically to create dope damage, set the requirement that dope damage be repaired with the function **objectstore::enable_damaged_dope_repair()**.

See the following in Chapter 2 of *ObjectStore C++ API Reference* for more information: **objectstore::enable_damaged_dope_repair()** and **objectstore::is_damaged_dope_repair_enabled()**.

Note that dope damage always occurs when unloading a DLL schema and it is always repaired, regardless of the setting of **objectstore::enable_damaged_dope_repair()**.

Schema Evolution

The **ossevol** utility can be used to evolve DLL (component) schemas.

Use the following keyword option:

ossevol <workdb> <schemadb> <evolvedb>+ [keyword_option]+

keyword_option ::= -dll_schema pthnames_of_component_schema

You can also use the following member functions of the class **os_schema_evolution**:

os_schema_evolution::augment_dll_schema_db_names

static void os_schema_evolution::augment_dll_schema_db_names (const os_charp_collection& dll_schema_db_names);

and

static void os_schema_evolution::augment_dll_schema_db_names (const char* c);

These two functions add the specified component schema database names to the list of component schema databases to be used for the evolution.

Exceptions

See Component Schema Exceptions in Appendix B of the *ObjectStore C++ API Reference*.

Index