ObjectStore®

Advanced C++ A P I User Guide

Release 6.3

PROGRESS SOFTWARE | Real Time Division

Advanced C++ API User Guide

ObjectStore Release 6.3 for all platforms, October 2005

September 2005

Contents

Preface

Purpose

The *Advanced C++ API User Guide* describes the way to use the C++ programming interface to ObjectStore to create database applications that use the more complex features of ObjectStore. This book supports ObjectStore Release 6.3.

This publication's companion volume, the *C++ API User Guide*, describes the basic features of the C++ programming interface to ObjectStore.

Audience

This book assumes that the reader is very experienced with C++ and with programming with ObjectStore in particular, especially with the information contained in the *C++ API User Guide*.

Scope

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

The Way This Book Is Organized

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

Some chapters of this book parallel chapters in the *C++ API User Guide*, providing a more advanced look at the ideas and features of ObjectStore, including persistence, transactions, threads and sessions, and schema evolution. The remainder of the chapters describe sophisticated features not generally used in more basic applications; these chapters describe integrity control, compaction, metaobject protocol, and dump/load.

Notation Conventions

This document uses the following conventions:

Convention	***Meaning***
`Courier`	`Courier font` indicates code, syntax, file names, API names, system output, and the like.
`Bold Courier`	**`Bold Courier font`** is used to emphasize particular code.
`Italic Courier`	*`Italic Courier font`* indicates the name of an argument or variable for which you must supply a value.
Sans serif	Sans serif typeface indicates the names of user interface elements such as dialog boxes, buttons, and fields.
Italic serif	In text, *italic serif typeface* indicates the first use of an important term.

Convention	*Meaning*
[ ]	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.

Progress Software Real Time Division on the World Wide Web

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

Technical Support

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

- You can send questions to realtime-support@progress.com. Remember to include your serial number in the subject of the electronic mail message.
- You can call the Technical Support organization to get help resolving problems. If you are in North America, call 1-781-280-4005. If you are outside North America, refer to the Technical Support Web site at www.progress.com/realtime/techsupport/contact.
- You can file a report or question with Technical Support by going to www.progress.com/realtime/techsupport/techsupport_direct.
- You can access the Technical Support Web site, which includes
  - A template for submitting a support request. This helps you provide the necessary details, which speeds response time.
  - Solution Knowledge Base that you can browse and query.
  - Online documentation for all products.
  - White papers and short articles about using Real Time Division products.
  - Sample code and examples.
  - The latest versions of products, service packs, and publicly available patches that you can download.
  - Access to a support matrix that lists platform configurations supported by this release.
  - Support policies.
  - Local phone numbers and hours when support personnel can be reached.

Education Services

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

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

Searchable Documents

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

Your Comments

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

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

Chapter 1
Advanced Persistence

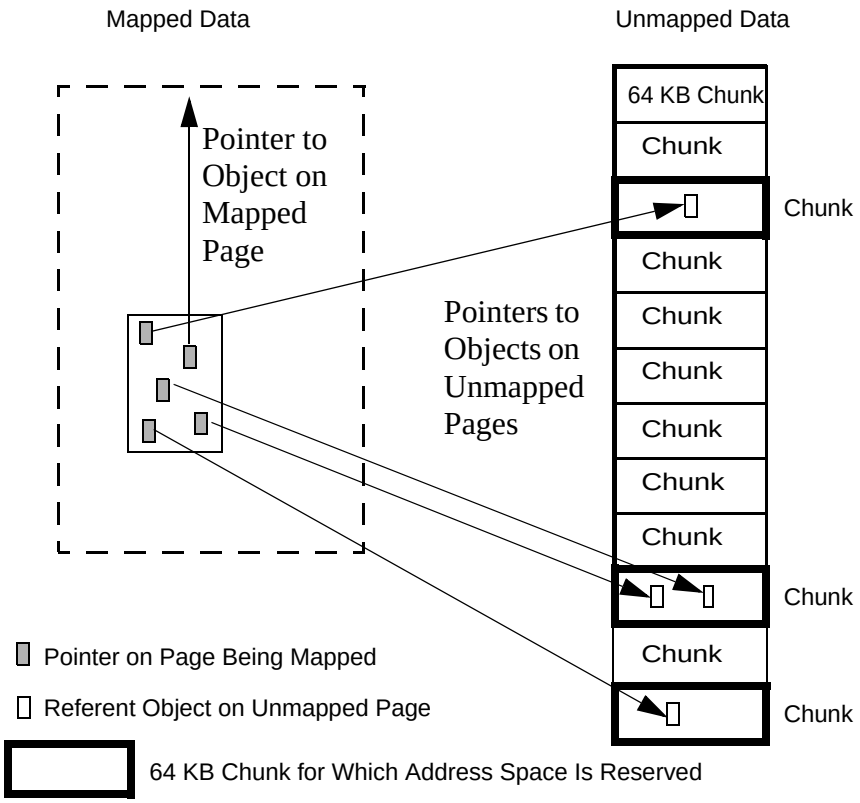
The information in this chapter augments Chapter 3, Programming with Persistent Storage, in the *C++ API User Guide*. The material is organized in the following manner:

Controlling Address Space Reservation

Address space is the space of all possible virtual memory addresses. Each address designates one byte of actual or potential virtual memory. When ObjectStore maps a page of data into virtual memory (see ObjectStore's Memory Mapping Architecture in the *C++ API User Guide*), it reserves address space for objects referenced by pointers on the mapped page. By default, ObjectStore unreserves this address space at the end of each transaction.

Address space is always reserved in 64 KB units. When address space is reserved for an object, ObjectStore actually reserves addresses for the entire 64 KB-aligned portion of memory containing the object. This document refers to a 64 KB-aligned

portion of memory for which address space is reserved as a *reservation chunk* or, simply, *chunk*.



On 32-bit platforms, if the pages used by a transaction refer to reservation chunks whose total size approaches the size of the current session's address space partition, you might have to control *address space reservation*.

Purpose of Address Space Reservation

Address space is reserved for a reservation chunk in preparation for the possibility of mapping pages in the chunk. If a page has a pointer to a reservation chunk, address space is reserved for the chunk when the page is mapped (unless space is already reserved for the chunk). If such a pointer is dereferenced, the page containing the referent is mapped at the appropriate offset from the location reserved for the chunk.

Controlling Address Space Reservation

As a transaction proceeds, the amount of address space reserved typically increases. As more pages are mapped, more chunks typically are referred to by mapped pages. As more chunks are referred to by mapped pages, more address space is reserved. By default, ObjectStore *releases* (that is, unreserves) all reserved address space at the end of each transaction.

Under some circumstances, the initial *fan-out* of a mapped page — the total amount of address space that must be reserved for all pointers on the page — can consume

all address space in the course of a single transaction. On 32-bit platforms, this might happen if the pages used by a transaction refer to chunks whose total size approaches the total size of the application's persistent storage region (or, for applications that use the sessions facility, the total size of the current session address space partition). When all address space is reserved, ObjectStore signals `err_address_space_full`.

Note

64-bit platforms are generally not at risk of running out of address space.

If you think a transaction might exhaust address space, first consider whether the transaction can be broken up into multiple shorter transactions, each of which consumes less address space. If this is not possible, you can control address space reservation in one of the following ways:

- Cluster data to reduce the amount of address space that must be reserved, as described in Clustering to Conserve Address Space on page 19.
- Release address space explicitly in midtransaction, before it is all reserved. (You can use `objectstore::is_unassigned_contiguous_address_space()` to determine if there is an unreserved address space portion of a given size.) For more information, see the next section.
- Handle `err_address_space_full` by releasing address space explicitly.
- Use soft pointers instead of certain hard pointers in your application schema to reduce the amount of address space reserved when pages containing the pointers are mapped into virtual memory. For more information, see Using ObjectStore Soft Pointers on page 22.

Midtransaction Address-Space Release Must Be User Controlled

When all address space is reserved, why does ObjectStore not automatically release some, just as it automatically evicts least recently used pages when the client cache is full? The answer concerns pointer validity. When pages are released from address space, all encached pointers to those pages become invalid because the virtual memory locations designated by the pointers can now be reserved for different data. Any automatic eviction scheme makes it difficult or impossible for the program to determine the pointers that have been rendered invalid.

With explicit address-space release, the program, in effect, specifies the pointers to be released, so there is no question about the pointers that have been rendered invalid and the program can take care not to use the pointers again. If the program subsequently needs pointers to the same objects, it knows to re-retrieve the pointers from the database.

Clustering to Conserve Address Space

Clustering can conserve address space by lowering the fan-out of a page of data when it is initially mapped into virtual memory — that is, by reducing the total amount of address space that ObjectStore must reserve for objects referenced by the pointers on the mapped page.

Good clustering not only groups related data (the *working data set*) on a minimal number of pages but also produces high locality of reference by grouping objects with the pointers that reference them. When such a page is mapped into virtual memory, the amount of address space that must be reserved for the page is minimal.

For information about using the ObjectStore API to cluster data, see Clustering in the *C++ API User Guide*.

Releasing Address Space

Use explicit release of address space to prevent reservation of all address space or to handle `err_address_space_full` (see Controlling Address Space Reservation on page 17). By default, all reserved address space is released automatically at the end of each transaction. You can release address space explicitly at other times as well, using one of the following:

- `objectstore::release_persistent_addresses()`; see Retaining and Releasing the Validity of All Pointers on page 31
- `os_address_space_marker`, discussed in this section

The class `os_address_space_marker` provides functions for

- Marking a point in the flow of execution
- Releasing all address space reserved since a specified marked point

To mark a point in the flow of execution of a program, create an instance of `os_address_space_marker` on the transient heap or the stack. The `os_address_space_marker` constructor has no arguments.

```
os_address_space_marker the_marker;
```

Call the member function `release()` on a specified marker to release all address space reserved since the creation of the marker (except address space retained by calls to `os_address_space_marker::retain()`, discussed later in this section, and any retained by `os_retain_address` objects. See Retaining the Validity of a Specified Variable on page 27). The function has no arguments and no return value.

```
the_marker.release();
```

Caution

Calling this function invalidates all pointers to the objects for which the released address space was reserved.

You can call `release()` on a marker repeatedly. After the first time you release a marker, releasing the marker releases the address space reserved since the last time the marker was released (except those retained by `retain()` or `os_retain_address`).

Example

Suppose that an application must process a large vector of pointers in a single transaction and that the pointers point to objects scattered among many different reservation chunks. Further suppose that after an element of the vector has been

processed, it is not used again in the transaction. To avoid exhausting address space, the application could release address space at various points during the traversal of the vector.

In the worst case, each vector element points to a different reservation chunk. In this case, the application should release address space after each `RES_LIMIT` element has been processed. `RES_LIMIT` is the size of the persistent storage region (or the current session's address space partition, for multisession applications) divided by 64 KB (minus the number of chunks occupied by addresses reserved before the marker and minus a few chunks for ObjectStore internal data).

If there are an average of 100 referents of the vector elements in each reservation chunk, the application can process 100 times as many elements before releasing address space.

Here is an example of releasing address space during a vector traversal:

```
  OS_BEGIN_TXN(tx1, 0, os_transaction::update)
{
  Foo *p = ...; // needed across release() calls

  os_address_space_marker the_marker;

  Foo *vec1[LARGE_NUMBER] = ...;
  for (int i = 0; i < vec_size; i++) {
    if ( (i-1) % RES_LIMIT == 0 )
      the_marker.release();
    process (vec1[i], p);
  }

}
OS_END_TXN(tx1)
```

Nesting Markers

Address space markers can be nested, which means that several markers can be in effect at the same time. Calling `os_address_space_marker::release()` on a marker releases the marker and all markers nested within it.

Retaining the Validity of Specified Pointers

The static function `os_address_space_marker::retain()` allows you to specify exceptions to the release of address space for pointers whose validity you want maintained across release boundaries. Here is its signature:

```
static void os_address_space_marker::retain(
  void *p,
  os_address_space_marker *marker = 0
);
```

If *`marker`* is `0`, the function retains the validity of *`p`* across calls to `os_address_space_marker::release()` (assuming that *`p`* is currently valid).

If *`marker`* is nonzero, the argument specifies a marker that you can use to release the address space occupied by *`p`*'s referent. In other words, the function retains the

validity of *p* across calls to `os_address_space_marker::release()`, except calls to `marker.release()`.

Caution

`marker` must not be nested within *p*'s *associated marker*, where a pointer's associated marker is the most deeply nested marker in existence when address space was reserved for the pointer's referent. If *`marker`* is nested within *p*'s associated marker, the call to `retain()` is ignored.

Deleting Markers

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

After the outermost marker is created, no more than 2^{30} minus 1 additional markers can be created before the outermost marker is deleted. This is true even if some or all of the inner markers are deleted.

Using Other Marker Functions

The class `os_address_space_marker` defines member functions for the following:

- Getting the most recently constructed marker that has not been deleted
- Getting a marker's nesting level
- Getting the marker created after a specified marker
- Getting the marker created before a specified marker
- Getting a pointer's associated marker

See the *C++ API Reference* for more information.

Using ObjectStore Soft Pointers

ObjectStore *soft pointers* provide an alternative to using regular pointers to persistent memory. Soft pointers reduce address space usage by deferring the reservation of address space for the soft pointer's referent until the soft pointer is dereferenced. See Controlling Address Space Reservation on page 17.

You can also use soft pointers to allow 32-bit applications to access persistent pointers stored by 64-bit applications, as well as to allow 64-bit applications to access pointers stored by 32-bit applications.

Dereferencing a soft pointer is somewhat slower than dereferencing a regular hard pointer. But the soft pointer facility uses a caching mechanism to increase soft-pointer efficiency. When a soft pointer is dereferenced, it resolves to a hard pointer. ObjectStore caches the dereferenced soft pointer along with the hard pointer to which it resolves. For this reason, a cached soft pointer is faster than a decached one because it really is just a hard pointer.

Also, when a page that contains cached soft pointers is reused, chunks of address space for the soft-pointer targets are immediately reserved.

Softness Is Application Relative

In a database, a soft pointer has the same format as a hard pointer. When one application stores a soft pointer in a database, another application subsequently can use the pointer as a regular hard pointer. Inversely, when one application stores a hard pointer in a database, another application can subsequently use the pointer as a soft pointer.

Specifying the Pointers That Are Soft

An application's (or DLL's) schema specifies those pointers that are treated as soft by the application (or DLL). For any pointer-valued data member in a database schema, the application's schema can specify the member's value type as a soft pointer class instead of a pointer type.

For example, suppose a database schema contains the class

```
class part
{
  employee* responsible_engineer;
  . . .
};
```

If your application accesses this database and you want the application to treat the values of `part::responsible_engineer` as soft pointers, define the application's class `part` in the following way:

```
class part
{
  os_soft_pointer<employee> responsible_engineer;
  . . .
};
```

These definitions of the class `part` are schema compatible. So although the application's definition of `part` and the database schema's definition of `part` differ, the application can access the database. When the application accesses an instance of `part`, it does not initially reserve address space for the part's responsible engineer (an instance of `employee`).

Because an application's schema specifies those pointers that are to be treated as soft, only values of data members can be treated as soft pointers. Top-level pointers or pointers in top-level arrays cannot be treated as soft.

To defer address-space reservation for referents of top-level pointers, replace the top-level pointers with instances of `os_Reference`. This class has an API just like the soft pointer API (see Soft Pointer API, following).

Soft Pointer API

Following is a brief description of the member functions provided by the soft pointer classes. Note that the *referent* of a hard or soft pointer is the object to which the pointer points or refers.

Constructors

The soft pointer classes provide constructors for

- Creating a soft pointer with the same referent as a given hard pointer (conversion constructor)
- Creating a null soft pointer (no-argument constructor)
- Copying a soft pointer (copy constructor)

Assignment operators

Overloadings of the assignment operator (`operator =()`) allow for modifying a soft pointer so that it has the same referent as a given hard or soft pointer.

Resolution functions

Before you dereference a soft pointer, you must *resolve* it, that is, retrieve a hard pointer with the same referent. Resolution is either implicit or explicit. Implicit resolution is supported by the dereference operator (`operator ->()`) and by a conversion operator, for conversion to a pointer to the referent type. Explicit resolution is supported by the member function `resolve()`. All these functions return a hard pointer with the same referent as `this`.

Other functions

The soft pointer classes also provide functions for comparing soft pointers with hard or soft pointers, as well as functions for hashing soft pointers and for dumping soft pointers to ASCII and loading soft pointers from ASCII. See the *C++ API Reference*.

Do not use the C++ `memcpy()` function or any means other than the soft pointer API to copy soft pointers. Using direct memory manipulation can cause the resulting copy to be unusable by ObjectStore.

Example

Because of the constructors, assignment operators, conversion operator, and `operator ->()` defined by the soft pointer classes, you often can use soft pointers just as you would use hard pointers. Following is an example of using a soft pointer, an instance of `os_soft_pointer`:

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

class employee
{
  static os_typespec *get_os_typespec();
  . . .
};

class part
{
  static os_typespec *get_os_typespec();
  . . .
  os_soft_pointer<employee> responsible_engineer;
  . . .
};

void f() {
  OS_ESTABLISH_FAULT_HANDLER
  objectstore::initialize();

  static os_database *db1 = os_database::open("/thx/parts");
  OS_BEGIN_TXN(tx1, 0, os_transaction::update)
```

```
    part *a_part = new(db1, part::get_os_typespec()) part;
    employee *an_emp =
      new(db1, employee::get_os_typespec()) employee;
    a_part->responsible_engineer = an_emp;

    printf("%d\n", a_part->responsible_engineer->emp_id);
  OS_END_TXN(tx1)

  db1->close();

  OS_END_FAULT_HANDLER
}
```

The member `responsible_engineer` is declared to be of type `os_soft_pointer<employee>`. The class name in angle brackets is the *referent type*. It indicates that values of `responsible_engineer` are soft pointers to instances of the class `employee`.

When a `part` is created,

```
part *a_part = new(db1, part::get_os_typespec()) part;
```

C++ implicitly calls the no-argument `os_soft_pointer` constructor, which constructs a null soft pointer as the value of the data member `responsible_engineer`.

When an `employee*` (`an_emp`) is assigned to the member `responsible_engineer`,

```
a_part->responsible_engineer = an_emp;
```

C++ invokes the soft pointer assignment operator, which makes the soft pointer point to the same object as `an_emp`.

When the value of `responsible_engineer` is dereferenced,

```
printf("%d\n", a_part->responsible_engineer->emp_id);
```

C++ invokes the soft pointer `->` operator, which resolves the soft pointer. C++ reapplies the `->` operator to the resulting hard pointer.

Note that implicit resolution is performed only by the `->` and conversion operators. Other operators, such as `[]` and `++`, do not perform implicit resolution and are not defined by the soft pointer classes.

Soft Pointer Classes and Macros

ObjectStore provides several soft pointer classes, but typically your code should refer only to instantiations of the class template `os_soft_pointer<T>`, such as `os_soft_pointer<employee>` in the preceding example. The template parameter is the referent type — the type of object referred to by the soft pointer.

Macro

`os_soft_pointer` is a macro, defined as `os_soft_pointer32` on 32-bit platforms and defined as `os_soft_pointer64` on 64-bit platforms. When you use `os_soft_pointer<T>`, therefore, you are actually using `os_soft_pointer32<T>` or `os_soft_pointer64<T>`.

Class templates

`os_soft_pointer32<T>` and `os_soft_pointer64<T>` define constructors, assignment operators, a conversion operator, `operator ->()`, and `resolve()`. Other

soft pointer operations are inherited from `os_soft_pointer32_base` or `os_soft_pointer64_base`.

Base classes

`os_soft_pointer_base<32>` and `os_soft_pointer_base<64>` define decache, hash, dump, and load functions. Comparison operators are defined in these base classes and in classes derived from them.

Nonclass referent types

If you use a soft pointer type whose referent type is not a class, you must call either the macro `OS_SOFT_POINTER32_NOCLASS()` or `OS_SOFT_POINTER64_NOCLASS()`, passing the referent type. This provides a definition for the template instantiation. For example, the following defines the class `os_soft_ptr32<float>`:

```
OS_SOFT_POINTER32_NOCLASS(float);
```

Implicit instantiation of the template defines `operator ->()`, which is invalid for nonclass parameters. Calling the macro defines an instantiation that does not provide `operator ->()`.

For soft pointers whose referent type is not a class, you resolve the pointer by calling `resolve()` or by relying on the conversion operator.

The ObjectStore API explicitly provides the following instantiations of `os_soft_pointer32<T>` and `os_soft_pointer64<T>`, using `OS_SOFT_POINTER32_NOCLASS()` and `OS_SOFT_POINTER64_NOCLASS()`:

- `os_soft_pointer32<void>` and `os_soft_pointer64<void>`
- `os_soft_pointer32<char>` and `os_soft_pointer64<char>`

Opening a Database Automatically

If a soft pointer refers to an object in a database that is not open, ObjectStore opens the database automatically when the object is accessed (unless the autoopen mode is `auto_open_disable`). The mode in which the database is opened is determined by the value of `objectstore::get_auto_open_mode()`.

Soft-Pointer Decaching

As mentioned in Using ObjectStore Soft Pointers on page 22, ObjectStore caches the targets of soft pointers to increase their efficiency. By default, ObjectStore does not automatically decache soft pointers on a page when that page is accessed again. When a page that contains cached soft pointers is reused, chunks of address space for the soft-pointer targets are immediately reserved.

If your application uses soft pointers and address space is an issue, you can force ObjectStore to decache soft pointers following the release of address space. When decaching is in effect and a page with soft pointers is reused, the soft pointers on the page are decached and the address space for their targets is not reserved.

To enable automatic, process-wide soft-pointer decaching , set the following environment variable to any nonnull value except `0`:

```
OS_ENABLE_DECACHE_SOFT_POINTERS_AFTER_AS_RELEASE
```

Alternatively, you can call `objectstore::set_decache_soft_pointers_after_address_space_release()`. To enable decaching, specify `true` (nonzero) as the argument; to disable decaching (the default), specify `false` (`0`). You should call this function immediately after creating the session to ensure the release of all address spaces that needs to be released.

To determine if decaching is enabled, call `objectstore::get_decache_soft_pointers_after_address_space_release()`. This function returns `true` if soft-pointer decaching is enabled and `false` otherwise.

You can also specify decaching for individual soft pointers, as described in `os_soft_pointer32_base::decache()` in the *C++ API Reference*. Note that decaching takes effect immediately after you call the `decache()` function. When process-wide decaching is enabled, however, soft pointers are decached the next time the page that contains them is reused, *following* the release of address space.

Soft-pointer decaching can degrade performance and should be enabled only if your application uses soft pointers and address space consumption is an issue. For more information about the impact of soft-pointer decaching on performance, see Soft-Pointer Decaching and Relocation on page 36.

For information about enabling soft-pointer decaching, see the following:

- `OS_ENABLE_DECACHE_SOFT_POINTERS_AFTER_AS_RELEASE` in *Managing ObjectStore*
- `objectstore::get_decache_soft_pointers_after_address_space_release()` in the *C++ API Reference*
- `objectstore::set_decache_soft_pointers_after_address_space_release()` in the *C++ API Reference*

Retaining the Validity of a Specified Variable

When a pointer to persistent memory is assigned to a transiently allocated variable, the value of the variable is valid only until the end of the top-level transaction in which the assignment was made. However, the ObjectStore API provides several ways to retain the validity of addresses to persistent memory across transaction boundaries, allowing you to assign a persistent address to a pointer in one transaction and dereference the pointer in another. The ways of doing this are as follows:

- Call `objectstore::get_retain_persistent_addresses()`, which globally reserves all persistent addresses. For more information, see Retaining and Releasing the Validity of All Pointers on page 31.
- Construct an `os_retain_address` object for a specific persistent object.
- Assign the address of a persistent object to an ObjectStore soft pointer.

`os_retain_address` vs. soft pointers

If you are interested in retaining the validity of a specific variable, you should consider using either `os_retain_address` or soft pointers. Both allow you to assign the address of a persistent object in one transaction and access it in another. One

difference between the two is that, unlike soft pointers, `os_retain_address` works by specially reserving the address space that holds the persistent object and not releasing it once the transaction ends. `os_retain_address` reserves address space in blocks of 64 KB. Depending on the amount of address space that is reserved, using `os_retain_address` for a number of large objects can result in problems in managing address space.

Although soft pointers do not reserve address space between transactions, assigning and dereferencing persistent memory is generally slower with soft pointers than with `os_retain_address`. `os_retain_address` allows you to reference the objects using C++ hard pointers. If you want to retain the addresses of a limited number of small objects, you should consider using `os_retain_address`.

The following sections discuss how to use `os_retain_address`. If you are interested in soft pointers, see Using ObjectStore Soft Pointers on page 22.

Using os_retain_address

You can cause such a variable's value to remain valid across top-level transaction boundaries with the class `os_retain_address`.

To cause a given variable's value to remain valid, create a stack-allocated instance of `os_retain_address`, passing to the constructor the address of the given variable:

```
document *the_doc_p = 0;
os_retain_address( (void**) &the_doc_p);
...
the_doc_p = (document*) (
    db1->find_root("documents_root")->get_value()
  );
```

As long as the associated instance of `os_retain_address` exists, ObjectStore retains the validity of the address that is currently or subsequently assigned to the variable. At the end of each top-level transaction, ObjectStore retains the validity of the value of the variable rather than unmapping it.

The instance of `os_retain_address` must be stack allocated; therefore, it is deleted automatically at the end of the block in which it was created. Because lexical transactions establish their own blocks, you do not typically call the `os_retain_address` constructor within a lexical transaction. The resulting instance would be deleted at the end of the transaction and would, therefore, be of no use in retaining validity across top-level transaction boundaries. (You could, however, use it to retain validity across calls to `objectstore::release_persistent_addresses()`.)

Note

Retaining the validity of a given address prevents release of a portion of address space at least 64 KB in size. See Controlling Address Space Reservation on page 17.

`os_retain_address` defines members for determining the address of an instance's associated variable and for changing an instance's associated variable. See `os_retain_address` in the *C++ API Reference*.

Example

Consider a pointer to a database entry point. Using an entry point typically involves looking up a root and retrieving its value, a pointer to the entry point. Frequently this pointer is assigned to a transiently allocated variable for future use. However, its use is limited because it typically does not remain valid in subsequent transactions.

Example of loss of pointer validity across transactions

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

void f() {
  OS_ESTABLISH_FAULT_HANDLER
  objectstore::initialize();

  static os_typespec part_type("part");

  part *a_part_p = 0;
  employee *an_emp_p = 0;

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

  OS_BEGIN_TXN(tx1,0,os_transaction::update)
    a_part_p = (part*) (
        db1->find_root("part_root")->get_value()
    ); /* retrieval */
    . . .
  OS_END_TXN(tx1)

  OS_BEGIN_TXN(tx2,0,os_transaction::update)
    an_emp_p = a_part_p->responsible_engineer; /* INVALID! */
    . . .
  OS_END_TXN(tx2)
  db1->close();
  OS_END_FAULT_HANDLER
}
```

Example of re-retrieving pointers in subsequent transactions

One way to ensure that the pointer remains valid is to re-retrieve the pointer in each subsequent transaction in which it is required.

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

f() {
  OS_ESTABLISH_FAULT_HANDLER
  objectstore::initialize();

  static os_typespec part_type("part");
  part *a_part_p = 0;
  employee *an_emp_p = 0;

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

  OS_BEGIN_TXN(tx1,0,os_transaction::update)
    a_part_p = (part*) (
      db1->find_root("part_root")->get_value()
    ); /* retrieval */
    . . .
  OS_END_TXN(tx1)
```

```
  OS_BEGIN_TXN(tx2,0,os_transaction::update)
    a_part_p = (part*) (
      db1->find_root("part_root")->get_value()
    ); /* re-retrieval */

    an_emp_p = a_part_p->responsible_engineer; /* valid */
    . . .
  OS_END_TXN(tx2)

  db1->close();

  OS_END_FAULT_HANDLER
}
```

Example of using os_retain_address

A convenient alternative is to use `os_retain_address`:

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

void f() {
  OS_ESTABLISH_FAULT_HANDLER
  objectstore::initialize();

  part *a_part_p = 0;
  os_retain_address( (void**) &a_part_p);
  employee *an_emp_p = 0;

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

  OS_BEGIN_TXN(tx1,0,os_transaction::update)
    a_part_p = (part*) (
        db1->find_root("part_root")->get_value()
    ); /* retrieval */

    . . .
  OS_END_TXN(tx1)

  OS_BEGIN_TXN(tx2,0,os_transaction::update)
    an_emp_p = a_part_p->responsible_engineer; /* valid */
    . . .
  OS_END_TXN(tx2)

  db1->close();

  OS_END_FAULT_HANDLER
}
```

Note that although you can use `a_part_p` from one transaction to the next without re-retrieving its value, you cannot use it between transactions. As always, you must be within a transaction to access persistent data.

Retaining and Releasing the Validity of All Pointers

To convert an existing application to use ObjectStore, it might be inconvenient to rewrite the code to re-retrieve pointers to persistent memory in each transaction or to use `os_retain_address` (see Retaining the Validity of a Specified Variable on page 27). A feature of ObjectStore enables you to retain the validity of all persistent addresses across transaction boundaries so that it is unnecessary to re-retrieve them or to use `os_retain_address` on a case-by-case basis.

The advantage of this feature is that code is easier to port to ObjectStore. The disadvantage is that ObjectStore might run out of address space if too much persistent data is referenced. This is because ObjectStore normally releases all reserved addresses at the end of each transaction and the use of this feature disables that release, which might leave less address space available for reservation in subsequent transactions. (See Controlling Address Space Reservation on page 17.) In addition, database access might be somewhat slower, particularly if multiple databases are referenced.

The static member function `objectstore::retain_persistent_addresses()` globally enables retaining persistent addresses. It has no arguments.

Releasing address space

The static member function `objectstore::release_persistent_addresses()` globally releases persistent addresses. After this function is called, all existing transient-to-persistent pointers are invalidated, except possibly those for which there is a corresponding instance of `os_retain_address` (see Retaining the Validity of a Specified Variable on page 27). If the argument to `release_persistent_addresses()` is `0` (the default), pointers for which there is a corresponding instance of `os_retain_address` are retained. If the argument to `release_persistent_addresses()` is nonzero, pointers for which there is a corresponding instance of `os_retain_address` are not retained.

You can determine whether persistent addresses are currently retained with `objectstore::get_retain_persistent_addresses()`, which returns nonzero for true and `0` for false.

Example: retaining persistent addresses

Following is an example of retaining addresses:

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

f() {
  OS_ESTABLISH_FAULT_HANDLER
  objectstore::initialize();

  os_database *db1;

    . . .
  person *p, *q;
    . . .
  objectstore::retain_persistent_addresses();

  OS_BEGIN_TXN(tx1,0,os_transaction::update)
```

```
    p = (person *) (db1->find_root("fred")->get_value());
    /* Now p is valid, and can be referenced normally. */
    p->print_info();

  OS_END_TXN(tx1)

  /* p cannot be dereferenced outside a transaction, but it */
  /* can be stored anywhere. */
  q = p;

  OS_BEGIN_TXN(tx1,0,os_transaction::update)

    /* If persistent addresses were not retained, we */
    /* could not do this without using references for p and q */
    q->print_info();

  OS_END_TXN(tx1)

  OS_ESTABLISH_FAULT_HANDLER
}
```

Persistent Pointers to Transient Objects

ObjectStore provides an API for managing persistent pointers to transient objects. For a description of this API, see the following sections in see the following sections of Chapter 2 of the *C++ API Reference*:

- `os_persistent_to_transient_pointer_manager`
- `objectstore::change_type()`
- `objectstore::release_cleared_persistent_to_transient_pointers()`
- `objectstore::set_persistent_to_transient_pointer()`

When an application accesses a persistent pointer to a transient object, it checks to see whether the pointer is null. If it is, the application is responsible for establishing a connection between the pointer and the object via an appropriate function. Then the application informs ObjectStore of the connection by calling the `objectstore::set_persistent_to_transient_pointer()` function. ObjectStore keeps track of these connections. ObjectStore clears the persistent pointers to transient objects when necessary so the transient pointers are not written to the database.

The application needs to derive an application-specific concrete class from `os_persistent_to_transient_pointer_manager`. The application is responsible for defining a `release()` function for the concrete class. This manager class and the `objectstore::release_cleared_persistent_to_transient_pointers()` function provide a way for ObjectStore and the application to clean up connections between persistent pointers and their associated transient objects. When an application calls the `objectstore::release_cleared_persistent_to_transient_pointers()` function, the appropriate user-defined `release()` function is run on each cleared pointer.

Your application can also call the `objectstore::clear_persistent_to_transient_pointers()` in order to immediately release transient storage.

Users should call `release_cleared_persistent_to_transient_pointers()` at a time when it is convenient to release transient storage that had been allocated in connection with calls to `set_persistent_to_transient_pointer()`. This time depends on the application, but after ending a transaction is a suitable time.

Example

Consider an example in which there are persistent objects of the `color` class whose data member, `gco`, is intended to point to a transient object based on the values of `r`, `g`, and `b`. The application code to establish and tear down this persistent-to-transient association would look something like this:

```
class color {
  ...
  private:
    int r, g, b;
    GUI_color_object *gco;

  public:
    GUI_color_object *get_gco();
    ...
};

class color_persistent_to_transient_pointer_manager: public /
os_persistent_to_transient_pointer_manager  {

  public:

// Application-specific code responsible for tearing down the
// persistent-to-transient association from the application's
// perspective and deleting the transient memory when appropriate.

  virtual void release(void * pointer);

// Application-added class-specific function for establishing
// the persistent-to-transient association.

  GUI_color_object * find_or_create_gco(int r, int g, int b);

  private:
// Whatever structures are necessary for the application's
// management of the persistent-to-transient association.

};
...

// somewhere accessible to the color object
color_persistent_to_transient_pointer_manager  color_pttp_mgr;

// Class function
  GUI_color_object * color::get_gco() {

// set up persistent-to-transient connection, if necessary
   GUI_color_object *transient_gco = gco;
  if (transient_gco == NULL) {
```

```
     transient_gco = color_pttp_mgr.find_or_create_gco(r, g, b);
    objectstore::set_persistent_to_transient_pointer(
      &gco,
      transient_gco,
      color_pttb_mgr
    );
  }
  return transient_gco;
}

// mainline application code
...
  OS_BEGIN_TXN(txn1, 0, os_transaction::update) {
    color *persistent_color_object;
    ...
    persistent_color_object->get_gco()->do_something();
    ...
  } OS_END_TXN(txn1)

     objectstore::release_cleared_persistent_to_transient_pointers();
  ...
```

ObjectStore's Relocation

Relocation refers to a set of tasks that the ObjectStore client performs on a persistent page, preparatory to making it accessible by the application or to copying the page to the server. These tasks require the client to examine and sometimes modify each field in an object, while preserving the logical content of the data in the field.

The following sections describe the two main types of relocation — inbound and outbound — and discuss how applications can influence relocation.

Inbound Relocation

Inbound relocation occurs when ObjectStore fetches a page from the server for access by the client. The ObjectStore client performs the following tasks to render the inbound page usable by the application:

- Create and reserve address space for objects referenced by pointers on the inbound page, as described in Controlling Address Space Reservation on page 17.
- Convert (or *swizzle*) pointers on the page from the encoded values they have in the database into address values that are meaningful to the application.
- Fill in C++ run-time dope, such as pointers (vptrs) to virtual function tables.
- Check the consistency between different pieces of metadata for the fetched page.
- If the current client is a different platform from that of the client that last modified the page, perform platform-related conversions to the data, such as byte-swapping.

Outbound Relocation

Outbound relocation occurs when a transaction has modified data that is subsequently committed or evicted from the client cache and therefore must be copied back to the server to be made persistent. The conversion work that occurs during outbound relocation includes:

- Pointer swizzling — converting the pointers on the page from their application format into the encoded format used by the database.
- Metadata checking.

Controlling Relocation by Clustering

Although application programmers have no direct control over relocation, they can reduce the need for relocation by clustering data. (For information about using the ObjectStore API to cluster data, see Clustering in the *C++ API User Guide*.) By clustering the working data set on a minimal number of pages, the application makes it more likely that those pages (or a subset of them) will stay in the cache over a sequence of transactions, thus reducing the number of page fetches and, as a side effect, reducing the number of times that the client must perform inbound relocation. Except for pages that are modified, a cached page requires no inbound relocation following the initial fetch.

Forward relocation

For pages that stay in the client cache over a sequence of transactions, ObjectStore gathers information about the pointers on each page during the previous transactions and about the address space that must be reserved for the pointers on the page. ObjectStore uses this information to perform what is called *forward relocation*, which consists of updating the cached page for subsequent access instead of having to subject the page to outbound relocation at the end of the previous transaction and inbound relocation in the current transaction. Only modified pages must undergo outbound relocation — in preparation for copying them back to the server and making them persistent. Forward relocation can be considered as an optimized combination of outbound relocation and inbound relocation that is applied to pages that stay encached between transactions.

Relocation optimization

In some cases, ObjectStore can skip forward relocation. This condition, known as *relocation optimization*, occurs when the chunks of address space that were reserved for a page in previous transactions are currently unreserved — that is, they have not been reserved by another page. In this situation, ObjectStore simply re-reserves the chunks targeted by the pointers on the page.

It is important to note that a cached page is eligible for relocation optimization only as long as the address-space chunks that were reserved for it in a previous transaction remain available — that is, unreserved — in succeeding transactions. (By default, ObjectStore unreserves chunks at the end of a transaction.) But if the application is at risk of running out of address space, ObjectStore will recycle previously reserved chunks that are not currently reserved, making them available for reservation by other pages. If a page has lost any of its previously reserved chunks to recycling, it is no longer eligible for relocation optimization and must undergo forward relocation.

One way to promote relocation optimization is to prevent the need for recycling by good clustering. As discussed in Clustering to Conserve Address Space on page 19, clustering can not only reduce the number of pages that an application will access over a series of transactions but also improve locality of reference by grouping objects with the pointers that reference them. Both of these effects of clustering can help reduce address-space consumption, thereby preventing the need for recycling.

Soft-Pointer Decaching and Relocation

As discussed in Soft-Pointer Decaching on page 26, you can use the following environment variable and function call to enable automatic soft-pointer decaching:

- `OS_ENABLE_DECACHE_SOFT_POINTERS_AFTER_AS_RELEASE`
- `objectstore::set_decache_soft_pointers_after_address_space_release()`

Decaching soft pointers is one way to force ObjectStore to unreserve chunks of address space that have been reserved for cached soft pointers, thus freeing up additional address space needed by the application. By default, the only soft pointers that ObjectStore decaches are those on pages that get forward relocation. When you enable automatic soft-pointer decaching, ObjectStore also decaches soft pointers on pages that would otherwise get relocation optimization. Such pages now get forward relocation.

Enabling soft-pointer decaching can impact performance, as follows:

- Once decached, a soft pointer takes longer to resolve than when encached, as described in Using ObjectStore Soft Pointers on page 22.
- Pages that are affected by automatic decaching lose the performance benefit of relocation optimization and instead must undergo forward relocation. Note that automatic decaching has *no* affect on pages that are eligible for relocation optimization and whose soft pointers have already been decached.

Before enabling soft-pointer decaching, you should weigh the benefit of unreserving address space against the cost in relocation performance.

Setting Data Fetch Policies

An ObjectStore application can control, for each cluster, the granularity of data transfers from the server to the client. When an application dereferences a pointer to an object that is not already resident in the client cache, ObjectStore retrieves from the server at least the page containing the object. The default behavior is to retrieve only the page containing the object. However, in some circumstances, retrieving additional pages can improve performance if the objects stored nearby in the database are likely to be referenced within a brief period of time.

Differences in granularity among fetch policies

ObjectStore has several *fetch policies* you can associate with a given cluster to control transfer granularity. Each fetch policy specifies the transfer granularity used when an object in the cluster needs to be transferred:

- `os_fetch_cluster` specifies that the entire cluster containing the desired object be fetched.
- `os_fetch_page` specifies that the page containing the desired object be fetched, along with zero or more of the pages that follow it in memory.
- `os_fetch_stream` specifies that a double buffering policy should be used to stream data from the referenced object's cluster. It is useful for special applications scanning large quantities of data.

Specifying a fetch policy

You specify the fetch policy for clusters, segments, or databases by using the member function `set_fetch_policy()`, declared as follows:

```
enum os_fetch_policy {
  os_fetch_page,
  os_fetch_stream,
  os_fetch_cluster,
};

static void set_fetch_policy(
  os_fetch_policy policy,
  os_int32 bytes
);
```

`os_cluster::set_fetch_policy()` sets the fetch policy for a specified cluster. `os_segment::set_fetch_policy()` sets the fetch policy for all clusters in a specified segment, including clusters created by the current process in the future. Similarly, `os_database::set_fetch_policy()` sets the fetch policy for all segments in a specified database. Finally, `objectstore::set_fetch_policy()` sets the fetch policy for all databases retrieved by the current process.

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

os_fetch_cluster Policy

Use the `os_fetch_cluster` policy when you are performing an operation that manipulates a substantial portion of a small cluster. Under this policy, ObjectStore attempts to fetch the entire cluster containing the desired page in a single client/server interaction if the cluster fits in the client cache without evicting any other data. If there is not enough space in the cache to hold the entire cluster, the behavior is the same as for `os_fetch_page`, with a fetch quantum specified by the *`bytes`* argument to the `set_fetch_policy()` function.

os_fetch_page Policy

If your database contains clusters larger than the client cache of your workstation or if your application does not refer to a significant portion of each cluster in the database, you should use the `os_fetch_page` fetch policy. This policy causes ObjectStore to fetch a specified number of bytes at a time (rounded up to the nearest positive number of pages), beginning with the page required to resolve a given object reference. Appropriate values for the fetch quantum might range from 4 KB to 256 KB or higher, depending on the size and locality of the application data structures.

```
os_cluster *text_cluster;

/* The text cluster contains long strings of characters */
/* representing page contents, which tend to be referred */
/* to consecutively. So tell ObjectStore to fetch them */
/* 16 KB at a time. */

text_cluster->set_fetch_policy (os_fetch_page, 16384);
```

os_fetch_stream Policy

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

This means that when you scan a cluster sequentially, after the first two transfers from the cluster, each transfer from the cluster replaces the data cached by the second-to-last transfer from that cluster. This way, the last two chunks of data retrieved from the cluster generally are in the client cache at the same time. And after the first two transfers, transfers from the cluster generally do not result in eviction of data from other clusters. This policy also greatly reduces the internal overhead of finding pages to evict.

```
os_cluster *image_cluster;

/* The image cluster contains scan lines full of pixel data, */
/* which we're about to traverse in sequence for image */
/* sharpening. Telling ObjectStore to stream the data from */
/* the server in 32 KB chunks gives us access to adjacent */
/* scan lines simultaneously and optimizes client/server traffic*/

image_cluster->set_fetch_policy (os_fetch_stream, 32768);
```

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

When the Fetch Quantum Is Too Large

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

Using Persistent Unions

Whenever you activate a member of a persistent union, including initially, you must make a function call indicating the active member. The function to call is `objectstore::set_union_variant()`. You can determine the active member of a union with `objectstore::get_union_variant()`. For more information about these functions, see the following sections of Chapter 2 of the *C++ API Reference:*

- `objectstore::get_union_variant()`
- `objectstore::set_union_variant()`

Union Overhead

Each union is associated with 16 bits of extra overhead in the form of internal schema information (minus savings due to optimization of certain cases involving union nesting). The use of unions also disables certain schema information storage optimizations. Consider benchmarking your application to determine if the actual memory savings outweigh the additional computational overhead.

Example

The following example shows the definition of a class with a union-valued member. `set_union_variant()` is called in the set functions for the member.

```
/*** example.cc ***/
#include "example.hh"

os_int32 main() {
OS_ESTABLISH_FAULT_HANDLER
  objectstore::initialize();
  os_database *db = os_database::create("example.db", 0666, 1);
  os_transaction::begin(os_transaction::update);
  kgraph *p_kg1 = new(db, kgraph::get_os_typespec()) kgraph();
  kgraph *p_kg2 = new(db, kgraph::get_os_typespec()) kgraph();
  if (!p_kg1->verify() || !p_kg2->verify()) return -1;
  p_kg1->set_int(5);
  if (!p_kg1->verify()) return -1;
  p_kg2->set_kg(p_kg1);
  if (!p_kg2->verify()) return -1;
  os_transaction::abort();
  db->destroy();
  return 0;
OS_END_FAULT_HANDLER
}

/*** example.hh ***/
#include <iostream>
#include <ostore/ostore.hh>

class kgraph {
private:
  union kg_union {
    kgraph* kg_kg;
    os_int32 kg_int;
  } kg;
```

```
  os_int16 kg_type;

public:
  kgraph()
  {
    kg_type = 0;
  }

  void set_kg( kgraph* new_kg)
  {
    kg_type = 1;
    if (objectstore::is_persistent(this))
      objectstore::set_union_variant(&this->kg,"kg_union",1);
    kg.kg_kg = new_kg;
  }

  void set_int( int new_int )
  {
    kg_type = 2;
    if (objectstore::is_persistent(this))
      objectstore::set_union_variant(&this->kg,"kg_union",2);
    kg.kg_int = new_int;
  }

  os_boolean verify()
  {
    os_int16 uv =
      objectstore::get_union_variant( &this->kg, "kg_union");
    if (kg_type != uv)
      cout << "union variant verification failed: expected "
        << kg_type << " but returned " << uv << endl;
    return (kg_type == uv);
  }

  static os_typespec *get_os_typespec();
};

/*** schema.cc ***/
#include <ostore/ostore.hh>
#include <ostore/manschem.hh>
#include "example.hh"

OS_MARK_SCHEMA_TYPE(kgraph);
```

Whenever either `kgraph::set_kg()` or `kgraph::set_int()` is called, the embedded call to `objectstore::set_union_variant()` records the currently active member object.

Setting Segment Reference Policies

Every segment has one of the two following reference policies:

- `export_id_access_required`: when a segment has this policy, certain reorganization facilities, such as compaction, can operate more efficiently on the segment. The cost of this policy consist of some extra cost to export objects in the

segment that might be referred to from other segments and some extra cost for traversing cross-segment pointers to the segment.

- `dsco_access_allowed`: this policy is the default and does not carry the costs described for `export_id_access_required`. Reorganizing the segment, however, might require scanning an entire database or group of databases, rather than scanning only that segment.

(These reorganization tools are provided in the current release or will be provided in subsequent releases.)

In addition to affecting reorganization, a segment's reference policy also affects garbage collection. The ObjectStore garbage collector automatically treats exported objects in `export_id_access_required` segments as garbage-collection roots. For garbage collection of `dsco_access_allowed` segments, you must identify the roots (the objects referred to by other segments) explicitly.

Setting and Getting Segment Reference Policies

To allow efficient operation of reorganization tools on a given segment, do both of the following:

- Specify the segment's reference policy as `objectstore::export_id_access_required`.
- Export every object in the segment that might be referred to by an object in a different segment. (See Exporting Objects on page 42.)

You specify a segment's reference policy when you create the segment.

```
enum objectstore::segment_reference_policy {
  export_id_access_required,
  dsco_access_allowed
};

os_segment* os_database::create_segment(
  objectstore::segment_reference_policy ref_policy
);
```

`dsco_access_allowed` refers to regular, nonexport ID access.

If you fail to export an object in an `export_id_access_required` segment, cross-segment pointers to the object are treated as illegal pointers (see `objectstore::set_null_illegal_pointers()` in the *C++ API Reference*).

You can get a segment's reference policy with the following function:

```
objectstore::segment_reference_policy
  os_segment::get_segment_reference_policy() const;
```

You can specify the default reference policy for newly created segments in a given database with the following:

```
void
os_database::set_new_segment_reference_policy(
objectstore::segment_reference_policy ref_policy);
```

`err_no_trans` is signaled if there is no transaction in progress at the time of the call.

You can also specify a default reference policy when you create a database. See `os_database::create()` and `os_database::open()` in the *C++ API Reference*.

You can get a database's default reference policy with the following:

```
objectstore::segment_reference_policy
  os_database::get_new_segment_reference_policy() const;
```

Exporting Objects

You export an object with `objectstore::export_object()`:

```
static os_export_id objectstore::export_object(
  void const * export_address
);
```

This function exports the top-level object pointed to by *`export_address`*. If *`export_address`* does not point to a top-level object, this function exports the top-level object containing the specified address. (An object is a top-level object if it is the entire object allocated by some call to `new`.)

Exporting an object assigns the object an ID (the object's *export ID*) that is unique within the object's segment. The export ID is used internally by ObjectStore. `export_object()` returns the exported object's export ID, an instance of `os_export_id`.

The database containing the exported object must be opened at the time of the call to `export_object()`. If the specified top-level object is not already exported, the database containing it must be opened for update and the current transaction must be an update transaction. If the object is already exported, calling this function is a read-only operation.

If you retrieve an export ID in a transaction that is later aborted, do not use the export ID after the abort. ObjectStore might reuse the export ID in a subsequent transaction.

For more information about `export_object()`, see `objectstore::export_object()` in the *C++ API Reference*.

Examining Export IDs

ObjectStore provides functions for manipulating export IDs in the following ways:

- Getting the export ID of a given object
- Getting all the export IDs in a given segment
- Getting the object associated with a specified export ID
- Copying, comparing, and assigning export IDs
- Getting the integer value of an export ID

There is also an export ID cursor class that supports traversal of all export IDs in a segment.

See `os_export_id` and `os_export_id_cursor` in the *C++ API Reference*.

Example

Following is an example that illustrates setting segment reference policies, exporting objects, and examining export IDs.

```
#include <iostream>
#include <ostore/ostore.hh>
#include <ostore/mop.hh>
#include "Widget.h"

int main(int argc, char** argv)
{
OS_ESTABLISH_FAULT_HANDLER
  objectstore::initialize();
  const char* db_name = argv[1];

  os_database* db = create_db(db_name, "widgets");

  show_db(db, "widgets");

  show_all_exports(db, "widgets");

  db->close();
  db->release_pointer();

OS_END_FAULT_HANDLER
}

os_database* create_db(const char* db_name, char* root_name)
{
  os_database* db = os_database::create(db_name, 0664, 1);

  os_transaction* txn =
    os_transaction::begin(os_transaction::update);

  // set the default segment reference policy for newly created
  // segments within the database.
  db->set_new_segment_reference_policy(
    objectstore::export_id_access_required
  );

  // create an export_id_access_required segment (per db-default)
  os_segment* seg1 = db->create_segment();

  // create an dsco_access_allowed segment
  os_segment* seg2 =
    db->create_segment(objectstore::dsco_access_allowed);
  // create an export_id_access_required segment explicitly
  os_segment* seg3 =
    db->create_segment(
      objectstore::export_id_access_required
    );

  Widget* widget1 =
    new (seg1, Widget::get_os_typespec()) Widget("X");

  // make sure that widget1 has an export_id assigned because it
  // will be the target of a cross-segment hard pointer.
  objectstore::export_object(widget1);
```

```
  Widget* widget2 =
    new (seg2, Widget::get_os_typespec()) Widget("Y");
  // seg2 doesn't require export-id access so the export isn't
  // required but it is ok to export locations within the
  // segment anyway.
  objectstore::export_object(widget2);

  Widget* widget3 =
    new (seg3, Widget::get_os_typespec()) Widget("Z");
  Widget** widgets =
    new (seg3, os_typespec::get_pointer(), 3) Widget*[3];

  widgets[0] = widget1;
  widgets[1] = widget2;
  widgets[2] = widget3;

  // set_value automatically exports the widgets array
  db->create_root(root_name)->set_value(widgets);

  txn->commit();
  delete txn;

  return db;
}

void show_db(os_database* db, char* root_name)
{
  os_transaction* txn =
    os_transaction::begin(os_transaction::read_only);

  os_database_root* widget_root = db->find_root(root_name);
  Widget** widgets = (Widget**)widget_root->get_value();
  widget_root->release_pointer();

  for (int i=0; i<3; i++) {

    os_export_id widget_export_id =
    objectstore::get_export_id(widgets[i]);

    if (widget_export_id.is_null()) {
      cout << "widget number " << i << " is not exported\n";
    }
    else {
      os_segment* widget_segment =
        os_segment::of(widgets[i]);
      cout << "widget number "
        << i << " is exported in segment "
        << widget_segment->get_number()
        << " with export id value "
        << widget_export_id.get_value() << "\n";
      widget_segment->release_pointer();
    }
  }

  txn->commit();
  delete txn;
}

void show_all_exports(os_database* db, char* root_name)
```

```
{
  os_transaction* txn =
    os_transaction::begin(os_transaction::read_only);
  os_segment_cursor segs(db);
  os_segment* seg;
  while (seg = segs.next()) {
    os_export_id_cursor eids(seg);
    os_export_id eid;
    while (!(eid = eids.next()).is_null()) {
      void* addr = seg->resolve_export_id(eid);
      os_type const* export_type = os_type::type_at(addr);
      cout << "export id value " << eid.get_value()
        << " in segment number " << seg->get_number()
        << " refers to ";
      if (export_type) {
        char* type_string = export_type->get_string();
        cout << "an object of type " << type_string << "\n";
        delete[] type_string;
      }
      else {
        cout << "an object of unknown type\n";
      }
    }
    seg->release_pointer();
  }

  txn->commit();
  delete txn;
}
```

Chapter 2
Advanced Transactions

This chapter is intended to augment Chapter 5, Transactions, of the *C++ API User Guide*. It includes descriptions of several advanced transaction concepts, particularly those pertaining to locks and locking. The information is organized in the following manner:

Reducing Wait Time for Locks

What can you do to reduce the overhead of waiting for locks? One application can reduce the waiting overhead for other concurrent applications by avoiding locking data unnecessarily and by avoiding locking data for unnecessarily long periods of time. This section describes several techniques for minimizing wait time.

Clustering

One way to help avoid locking data unnecessarily involves the use of clustering. Suppose that during a given transaction, an application requires `object-a` but not `object-b`. If the two objects are clustered onto the same page, they are both locked, preventing other processes from accessing both objects until the end of the transaction. In contrast, by clustering `object-b` in a different object cluster or segment from `object-a`, you guarantee that the objects will be on different pages. So if you use page-level locking granularity, the objects are not locked together.

Locking Granularity

By default, each page is locked as it is faulted on; the default locking granularity is page level. You can override this default with the `set_lock_option()` member of

`os_cluster`, `os_segment`, `os_database`, or `objectstore`, specifying cluster-, segment-, or database-level granularity. See the *C++ API Reference*.

Overriding the default avoids the overhead of a separate page fault for each page locked and can reduce server communication. It can also lock data unnecessarily and increase wait time.

Transaction Length

One way to avoid locking data for unnecessarily long periods of time is to make nonnested transactions as short as possible while still guaranteeing that persistent data is in a consistent state between transactions.

The disadvantage of using shorter transactions is that it can mean using a greater number of transactions. This can increase network overhead because each transaction commit requires the client to send a *commit message* to the server. Nevertheless, this extra network overhead is often outweighed by the savings from shorter waits for locks to be released.

Multiversion Concurrency Control (MVCC)

Read-only transactions can use *multiversion concurrency control*, or MVCC. When you open a database using MVCC, you can perform nonblocking reads of the database allowing another ObjectStore application to update the database concurrently, with no waiting by either the reader or the writer. See Multiversion Concurrency Control (MVCC) on page 51 for additional information.

Abort-Only Transactions

The locking restrictions are relaxed somewhat for abort-only transactions. With abort-only transactions, the client does not get write locks for any pages that are written during an abort-only transaction. The client does get read locks for all pages it reads or writes. This lock relaxation is another method of reducing wait time.

Lock Timeouts

Lock timeouts provide the ability to limit wait time and to abort if limits are exceeded. You can set a timeout for read- or write-lock attempts, to limit the amount of time your application will wait. 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. `set_readlock_timeout()` and `set_writelock_timeout()` are members of the `objectstore`, `os_database`, and `os_segment` classes, which are all described in Chapter 2 of the *C++ API Reference*.

Nested Transactions

For a number of reasons, it is useful to allow transactions to be nested. For example, suppose one transaction is required to hide intermediate results. This also allows rollback of persistent data to its state as of the beginning of the transaction. But

suppose you would like to be able to roll back persistent data to its state as of some point after the beginning of this transaction. To allow this, you can use a nested transaction that starts at this later point.

In addition, allowing nested transactions means that a routine that initiates a transaction can be called from both inside and outside a transaction.

When a nested transaction is aborted, persistent data is rolled back to its state as of the beginning of that nested transaction.

Caution

After a nested transaction aborts, do not use persistent values retrieved during the aborted transaction. See Locking and Nesting, following.

Locking and Nesting

Locks acquired during a given nested transaction are released upon whichever of these conditions comes first:

- Abort of the given nested transaction
- Abort of a transaction within which the given transaction is nested
- Commit of the top-level transaction within which the given transaction is nested

This means that after a nested transaction aborts, you should not use persistent values retrieved during the nested transaction because they might not be consistent with values retrieved later in the same top-level transaction. Consider, for example, the following nested transactions:

```
start outer txn
  start nested txn
    read persistent var x
  abort nested txn
  start nested txn
    read persistent var x
  commit nested txn
commit outer txn
```

The two read operations performed on `x` might not have the same result. Another process can change the value of `x` in the interval between the abort of the first nested transaction and the next read of `x`.

Abort-Only Transactions

An update transaction nested within a read-only transaction is known as an *abort-only* transaction. In an abort-only transaction, you can write to persistent data, but you must end the transaction with an abort.

Abort-only transactions allow you to write to databases that are

- Read-only
- Protected against writes
- MVCC opened

You must end an abort-only transaction by calling `abort()` or `abort_top_level()`. ObjectStore signals `err_commit_abort_only` if you try to commit an abort-only transaction.

With an abort-only transaction, the client does not get write locks for any pages written during the transaction. Thus there can be multiple concurrent abort-only writers to a database. The client does get read locks for all pages it reads or writes.

Deadlock

ObjectStore sometimes aborts a transaction automatically because of deadlock. A simple deadlock occurs when one transaction holds a lock on a page that another transaction is waiting to access, while at the same time this other transaction holds a lock on a page that the first transaction is waiting to access. Neither process can proceed until the other does. See Simple Deadlock Scenario on page 63. There are other, more complicated forms of deadlock that are analogous.

Deadlock Victim

ObjectStore has a deadlock detection facility that breaks deadlocks, when they are detected, by aborting one of the transactions involved in the deadlock. By aborting one transaction (the *victim*), ObjectStore causes the victim's locks to be released so that other processes can proceed.

You can control the way ObjectStore chooses a victim with `objectstore::set_transaction_priority()` (see Chapter 2 of the *C++ API Reference*) and the `Deadlock Victim` server parameter (see Chapter 2 of *Managing ObjectStore*).

Automatic Retries Within Lexical Transactions

When a lexical transaction (one specified with the transaction statement macros) is aborted because of a deadlock, the system retries the aborted transaction automatically.

In the event that the transaction is repeatedly aborted by the system, the retries continue until the maximum number of retries has occurred. This maximum for any transaction in a given process is determined by the value of the static data member `os_transaction::max_retries`. You can retrieve the value of this member with `os_transaction::get_max_retries()`.

```
static os_int32 get_max_retries() ;
```

Changing the maximum number of retries

The default value of `max_retries` is `10`. You can change the value of `max_retries` at any time with `os_transaction::set_max_retries()`.

```
static void set_max_retries(os_int32) ;
```

The change remains in effect only for the duration of the process and is invisible to other processes.

Consequences of Automatic Deadlock Abort

When a transaction is aborted by the system, its changes are undone. However, only persistent state is rolled back. Transient state is not undone, and any form of output that occurred before the abort is not, of course, undone. Sometimes it is a good idea

to perform output outside a transaction, but other times this might not be the best approach.

Deadlocks in Dynamic Transactions

Dynamic transactions that are aborted because of deadlock are not retried. If you want to retry a dynamic transaction aborted because of deadlock, you must do so explicitly by handling the exception `err_deadlock`. Call `os_transaction::abort_top_level()` from within the handler.

See Using Dynamic Transactions in Chapter 5 of the *C++ API User Guide* for more information about dynamic transactions.

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 MVCC.

If a transaction does both of the following, you should open the database for MVCC.

- Only performs read access on a database
- Does not require a view of the database that is completely up to date, but can instead rely on a snapshot of the data

You can open the database for MVCC with members of the class `os_database` (see MVCC API on page 52).

No Waiting for Locks

If an application has a database opened for MVCC, it never has to wait for locks to be released in order to read the database. Reading a database opened for MVCC also never causes other applications to have to wait to update the database; see the example MVCC and the Simple Waiting Scenario on page 63. In addition, an application never causes a deadlock by accessing a database it has opened for MVCC. See the example MVCC and the Simple Deadlock Scenario on page 64.

Snapshots

In each transaction in which an application accesses a database opened for MVCC, the application sees what it would see if viewing a snapshot of the database taken sometime during the transaction. If an application accesses databases that have been opened for multidatabase MVCC it will see snapshots of those databases taken at the same moment in time.

This snapshot has the following characteristics:

- It is internally consistent.

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

Performing a checkpoint is often a good way to update an application's snapshot. See Performing Transaction Checkpoints on page 56.

Accessing Multiple Databases in a Transaction

You can open databases for MVCC in either single-database or multidatabase mode. When an application reads a database opened for single-database MVCC, the snapshot it sees is not necessarliy consistent with other databases accessed in the same transaction (although it is always internally consistent).

When an application reads databases opened for multidatabase MVCC, the snapshot is a consistent, aggregate view of all data items in all the databases at the time the snapshot was taken. To ensure that an application maintains a consistent view of all data in the database open for multidatabase MVCC, the application should handle the `err_mvcc_incoherent` exception as described in Handling err_mvcc_incoherent on page 54

Serializability

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

MVCC API

You open a database for single-database MVCC with one of the following members of `os_database`:

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

You open a database for multidatabase MVCC with one of the following members of `os_database`:

```
void open_multi_db_mvcc();
static os_database* open_multi_db_mvcc(const char* pathname) ;
```

It is valid to open MVCC databases by following cross-database pointers.

After you open a database for MVCC, MVCC is used for access to that database until you close it. If the database is already opened, but not for MVCC, `err_mvcc_nested` is signaled. If you try to perform write access on a database opened for MVCC, `err_opened_read_only` is signaled.

You can determine if a database is already opened for single-database or multidatabase MVCC with the following members of `os_database`:

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

Returns nonzero if the database is opened for either single-database MVCC or multidatabase MVCC, and `0` otherwise.

```
os_boolean is_open_single_db_mvcc() const;
```

Returns nonzero if the database is opened for single-database MVCC, and `0` otherwise.

```
os_boolean is_open_multi_db_mvcc() const;
```

Returns nonzero if the database is opened for multidatabase MVCC, and `0` otherwise.

You can update an application's snapshot of a database with `os_transaction::checkpoint` (see Performing Transaction Checkpoints on page 56) or by closing and then reopening the database.

MVCC and the Transaction Log

Although MVCC can cause ObjectStore to, in effect, take a snapshot of an entire database, the implementation actually copies data only when needed, on a page-by-page basis. Moreover, making the copy simply amounts to retaining the page in the transaction log. See Logging and Propagation on page 55.

In the absence of MVCC, updated pages from committed transactions are propagated from the log to the database on a periodic basis. But with MVCC, updated pages are held in the log as long as necessary, so that the corresponding page in the database is not overwritten and can be used as part of the MVCC snapshot.

Caution

This means that long transactions that use MVCC can cause the log to become very large.

Conflict detection

MVCC determines whether a page must be held in the log based on the following notion of conflict. From the time a conflict is detected in a given transaction, propagation is delayed for subsequently committed data until the given transaction ends.

A conflict occurs when one of the following happens:

- A process tries to read a page in a database it has opened for MVCC and another process has the page write locked.
- A process tries to write a page in a database and another process that has the database opened for MVCC has the page read locked.

In both of these cases, both processes proceed; no one is blocked. The transaction performed by the MVCC process is placed just before the conflicting update transaction in the serialization order. This is effectively when the snapshot is taken. See MVCC Conflict Scenario on page 64.

Under some circumstances, the ObjectStore server might decide to hold a page in the log in anticipation of a conflict, even if none has actually occurred.

Handling err_mvcc_incoherent

ObjectStore throws the `err_mvcc_incoherent` exception when it cannot guarantee a consistent snapshot of all databases currently opened for multidatabase MVCC, as in any of the following situations:

- Your application calls an overloading of `os_database::open_multi_db_mvcc()` to open a database explicitly for multidatabase MVCC during a transaction.
- Your application follows a cross-database pointer, the target database is not already open, and the application is in multidatabase MVCC autoopen mode.
- Your application loses its connection to the server for a database that is opened for multidatabase MVCC, re-establishes the connection, and then fetches some of the database's data.
- Server-to-server communication fails while a snapshot of databases on multiple servers is being taken, and then your application fetches some of the database's data from one of the servers involved.

If an `err_mvcc_incoherent` exception occurs during a dynamic transaction, a handler should abort the current transaction and then retry it. If the exception occurs in a lexical transaction, the transaction will be automatically retried. You can also provide your own transaction handler inside a lexical transaction in place of the automatic retry. If you attempt to commit either a dynamic or lexical transaction in which the `err_mvcc_incoherent` exception has occurred, an `err_cannot_commit` exception will be thrown. If the application proceeds with the current transaction instead of aborting it, the application may see an inconsistent view of the data, producing incorrect results in the transaction. Be sure to verify that any inconsistency does not compromise the accuracy of your application's results.

When ObjectStore throws the `err_mvcc_incoherent` exception, the database whose opening caused the exception remains open. Therefore, the retry should not reopen the database. Reopening the database will increase the open count, possibly defeating a later attempt to close the database.

Performance Note

Although MVCC can eliminate locking delays that can occur in regular, non-MVCC transactions, they can still cause an application to wait for the resolution of a callback. (See Ownership and Locks in Chapter 1, *Managing ObjectStore* for information about callbacks.) In most cases the resulting delay is only a matter of milliseconds. Such a delay can occur under the following conditions:

- An MVCC client reads a page for which an update client has write ownership, or an update client acquires a write lock on a page for which an MVCC client has read ownership.
- A snapshot has not already been taken.
- The other client does not have a lock on the page that the server knows about.

In this situation, a callback occurs to check whether there is a conflict that requires a snapshot to be taken. There is no conflict if the page is not locked and thus can be

called back. Delays related to callbacks can occur in either single-database or multidatabase MVCC.

In addition, when a multidatabase MVCC snapshot is being taken concurrently with the commit of an update transaction, and the intersection of the snapshot's data with the update transaction's written data spans more than one server, a delay may occur while the servers determine whether the update is or is not visible in the snapshot. In most cases, this delay is unnoticeable, but in the worst case it may take a few seconds. This delay can slow either the MVCC client or the update client and is not under user control. It depends on network communication delays or the load on the server. Of course, if all the multidatabase MVCC databases opened by a given client are hosted by one server, a delay can never arise.

Logging and Propagation

The ObjectStore *transaction log*, as with the log in any database system, is used to ensure fault tolerance and to support the functionality involved in transaction aborts. The log is stable storage (that is, disk storage) used to keep temporary copies of data on its way to the database from the client cache.

Transaction Logging

Data is recorded in the log before being written to the database (with certain exceptions — see the following paragraphs) and is not removed from the log until some time after the transaction sending it has committed. That way, if a failure occurs in the middle of moving a transaction's data to the database (for example, because the network crashes or the power fails on the server host), the data is nevertheless safely in the log and can be moved to the database in its entirety during recovery.

If a failure occurs before or during the recording of a transaction's data in the log, the transaction is considered to have aborted, and the data is never written to the database (and similarly, if the transaction aborts because of deadlock or a call to `abort()`, the data is never written to the database).

New data whose creation results in the use of new disk sectors is handled differently. This data is sometimes moved directly to the database, and sufficient information is maintained on stable storage to effectively remove the data from the database if the creating transaction aborts. For new sectors and segments, this undo information is kept in the database itself; for new databases, the information is stored in the log as an undo record.

Propagation

During normal operation, the ObjectStore server moves, or *propagates*, data from the log to the database on a periodic basis. The server keeps track of what has been propagated and always knows whether the latest committed version of any given sector is to be found in the log or in the database. That way, when clients request data from the server, the server can send the sector's most up-to-date version.

Controlling propagation

You can control how often propagation occurs with the ObjectStore server parameter `Propagation Sleep Time`; the default is every 60 seconds. This determines the time between propagations, except when the server temporarily deems it necessary to propagate on a more frequent basis. By default, the server increases the propagation rate when there are more than 8192 sectors waiting to be propagated. You can override the default of 8192 with the server parameter `Max Data Propagation Threshold`. The server also increases the propagation rate to empty out a log record segment.

You can control the amount of data propagated each time with the server parameter `Max Data Propagation Per Propagate`. For propagates that consist of a single disk write (that is, propagation of data that is contiguous in the database), this specifies the number of sectors to propagate (the default is 512).

For information on these and other server parameters, see Chapter 2, Server Parameters, in *Managing ObjectStore*.

Performing Transaction Checkpoints

By performing a checkpoint operation on a transaction, you can commit modified data from a top-level transaction without incurring some of the usual overhead of ending a transaction and starting a new one. A checkpoint operation

- Commits the current top-level transaction.
- Immediately starts a new transaction.
- Immediately acquires read locks on all or most persistent objects that were locked before the checkpoint. This operation is generally less expensive than acquiring the locks incrementally, as happens without checkpoint in an explicitly started transaction.
- Retains the validity of pointers to persistent memory across the checkpoint boundary. Note that address space is not released across checkpoint boundaries. See Controlling Address Space Reservation on page 17.
- Refreshes the application's view of any database opened for MVCC. See Multiversion Concurrency Control (MVCC) on page 51.

This is useful when you are making modifications to a database and you want to periodically commit your changes but continue updating the database without intervention. This might be the case, for example, when you are bulk-loading new data into a database.

Checkpointing a Transaction

To checkpoint a transaction, call `os_transaction::checkpoint()`. Thgis function has two overloadings. The no-argument overloading is used to perform a checkpoint on the current transaction, and `os_transaction` overloading performs a checkpoint on the specified transaction.

Note that, checkpoints within a transaction differ from conventional checkpoints. In this checkpoint, an application might not have all the locks after the checkpoint that it had before the checkpoint.

Like transaction commit and abort operations, a checkpoint operation is not thread safe. Applications must ensure that other threads in the same session do not access persistent memory during a checkpoint operation.

For more information about the `checkpoint()` function, see `os_transaction::checkpoint()` in the *C++ API Reference*.

Locking and Checkpoints

If another client is waiting for a write lock on a persistent object that was locked in your transaction, you lose that lock when you checkpoint the transaction. As long as another client is not waiting for a write lock on an object that was associated with your transaction, you reacquire as read locks any locks you had before the checkpoint.

If there were any write locks before the checkpoint, ObjectStore changes them to read locks or gives them to any clients waiting for those write locks. Consequently, you might have to wait for locks or you might get a deadlock when you try to update the database again.

Pointer Validity

After the checkpoint, you do not have to start from a root object to set up your access to objects. Your application's access to objects can be the same before and after the checkpoint.

Support for the XA Standard for Transaction Processing

ObjectStore supports X/Open's transaction demarcation protocol (known as XA).

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.

In the DTP model, transaction demarcation is controlled by a *transaction manager*.

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 can be multiple ObjectStore databases or incompatible databases, such as ObjectStore and Sybase.

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.

An ObjectStore client acts as a *resource manager*. Resource managers must be registered with the transaction manager.

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.

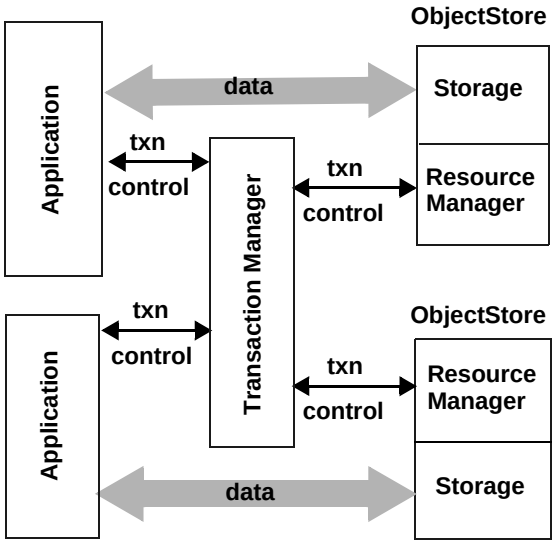
XA transactions are identified by XIDs (universally unique identifiers for global transactions). A branch has a globally unique ID composed of the global transaction ID and a branch ID.

XA transactions are always of the type `update`.

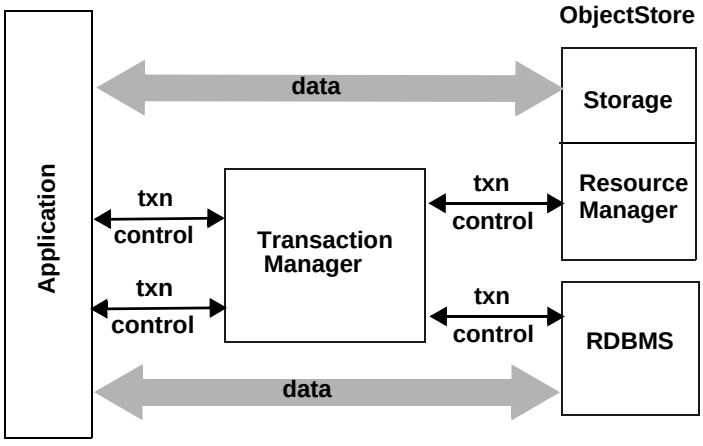
XA is also integrated into the OMG Object Transaction Service (OTS), which is based on DTP.

An application that uses a transaction manager must use the transaction manager'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 transaction manager manages the process, using the XA interfaces to roll back web 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 example below uses the Orbix OTS as the transaction manager and describes the way to register a resource manager by passing a set of arguments to the `Encina::Server::RegisterResource` function.

Information for the following parameters is required

Parameter	*Description*
`xaSwitchP`	Pointer to a data structure of type `xa_switch_t`. Specify the symbol `ObjectStore_xa_switch` for ObjectStore resource managers.
`openString`	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`.

Initialization example

The following example shows 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

After you register 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, the exception `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 it starts a transaction.

Following 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 they are 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.
- Multiserver 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.

Transaction Locking Examples

The following examples illustrate some of the locking situations described in this chapter.

Simple Waiting Scenario

If one transaction reads a page and then another transaction reads the same page, the second transaction is not blocked. However, if the second transaction tries to write to the page, it must wait until the first transaction commits.

Transaction 1	*Transaction 2*
`Read P`	
	`Read P`
	`Write P: BLOCKED`
`Commit`	

So the actual schedule of operations looks like the following:

Transaction 1	*Transaction 2*
`Read P`	
	`Read P`
`Commit`	
	`Write P (succeeds)`

Simple Deadlock Scenario

In the schedule below, Transaction 2 attempts to write `P1` but cannot proceed until Transaction 1 completes and releases its read lock on `P1`. However, Transaction 1 cannot proceed until Transaction 2 completes and releases its lock on `P2`. Because neither transaction can proceed until the other does, the result is a classic deadlock scenario. ObjectStore chooses Transaction 1 as victim and aborts it, whereupon Transaction 2 can proceed.

Transaction 1	*Transaction 2*
`Read P1`	
	`Read P1`
	`Read P2`
	`Write P2`
	`Write P1: BLOCKED`
`Read P2: BLOCKED — DEADLOCK`	
`Abort`	
	`Write P1 (succeeds)`

MVCC and the Simple Waiting Scenario

If one transaction reads a page of a database it has opened for MVCC and then another transaction attempts to update the same page, the second transaction is not blocked. Compare this with the Simple Waiting Scenario on page 62.

MVCC Transaction 1	*Update Transaction 2*
`Read P`	
	`Read P`
	`Write P: NOT BLOCKED`

MVCC and the Simple Deadlock Scenario

In the schedule below, Transaction 2 writes `P1` without waiting; it can proceed before Transaction 1 completes and releases its read lock on `P1` because Transaction 1 has the database containing the page opened for MVCC. Similarly, Transaction 1 can proceed before Transaction 2 completes and releases its lock on `P2`. Without MVCC, deadlock would have resulted. See the Simple Deadlock Scenario on page 63.

MVCC Transaction 1	*Update Transaction 2*
`Read P1`	
	`Read P1`
	`Read P2`
	`Write P2`
	`Write P1: NOT BLOCKED`
`Read P2: NOT BLOCKED`	

MVCC Conflict Scenario

MVCC and `update` conflict because `update` writes something (`A`) which is being read by MVCC. Therefore, all pages updated by `update` must be retained in the log so that MVCC can see the old copies of these pages.

MVCC Transaction 1	*Update Transaction 2*
`Read A`	
	`Read A, B, C, D, E`
	`Write A, B, C, D, E`
	`Commit`
`Read B (old)`	

Chapter 3
Multithread and Multisession Applications

ObjectStore supports the use of multiple threads within a client application. The key to developing a successful multithreaded application with ObjectStore lies in choosing

- The right assignment of operations to threads
- The right assignment of threads to *sessions*

These topics are discussed in the first two sections of this chapter:

The rest of the chapter describes using the sessions facility and consists of the following sections:

For more information

For information about using the ObjectStore API to set up thread locking, see Threads Support in Chapter 1 of the *C++ API User Guide*. The `os_session` class provides user access to the sessions facility. For information about the member functions this class, see `os_session`, in the *C++ API Reference.*

What Are Sessions?

A *session* provides a context for the execution of specified portions of one or more threads. A session is, in many respects, exactly like an ObjectStore client. Each session is a subclient of an ObjectStore client and has its own

- Cache
- Server and cache manager connections
- Communications segment (commseg)
- Portion of the persistent storage region of the current process (*address space partition*)

If you do not use the sessions facility explicitly, all your application's threads execute within a single session. If you use the sessions facility, concurrent threads can be in different sessions or in the same session.

Threads in Different Sessions

If you use the sessions facility,

- ObjectStore synchronizes access to persistent data by threads in different sessions automatically.
- Threads in different sessions can run in different transactions concurrently.
- Threads in different sessions can operate against the same database opened for different types of access — MVCC, update, or read-only access — concurrently.

ObjectStore performs the same kind of automatic lock management whether different sessions or different clients perform concurrent access to persistent data. Therefore, threads in different sessions are synchronized using the ObjectStore concurrency control facility and can achieve the same degree of concurrency that separate ObjectStore client processes can.

Threads in the Same Session

Threads in the same session can use either global or local transactions.

Global transactions

With global transactions, threads in the same session can participate in the same transaction. When one thread starts a global transaction, all threads in that thread's session enter that transaction. When one thread ends a global transaction, all threads in that thread's session leave that transaction.

For threads in the same global transaction, ObjectStore thread locking prevents two threads from operating within the ObjectStore run time at the same time (but see Thread Locking and Collections on page 67). So, if you run multiple threads in the same global transaction, you are responsible for synchronizing the threads' access to persistent data and for ensuring that the threads perform no persistent access during a commit or checkpoint.

Local transactions

With local transactions, the transactions of different threads in the same session are serialized. When one thread starts a local transaction, other threads in the same

session are blocked until the transaction commits. So if you run multiple threads in the same session and use local transactions, the threads are automatically synchronized. But the degree of concurrency among the threads is lower than it would be if the threads were in different sessions.

Thread Locking and Collections

This section describes ObjectStore thread locking for collections. ObjectStore thread locking is used for threads executing within the same global transaction.

ObjectStore thread locking ensures that no two threads execute within the ObjectStore run time at the same time, with the exception of parts of the collections run time. Because of automatic thread locking, no two threads can create a query or execute a query against the same collection at the same time. Automatic thread locking, however, does not prevent different threads from performing other types of collection access at the same time.

Therefore, for threads in the same global transaction, you are responsible for synchronizing all other forms of access to ObjectStore collections, such as insertion, deletion, and traversal with cursors.

Because of blocking from thread locking, an unoptimized query might not perform as well as an equivalent explicitly coded traversal of the collection.

Designing Multithreaded Applications

Many applications take streams of requests from other processes. For each request, they perform a requested operation and send a reply. For example, many application servers accept, process, and reply to requests from thin client processes running on web-server hosts. These applications benefit from multithreading because the requests represent discrete, independent units of work that can be performed in any order and can, therefore, be handled by multiple threads.

An effective design for these applications groups requests according to the style of transaction management appropriate for each request (see Transaction Management Styles). With this design, the application creates a session for each style of transaction management and one or more threads for each session. A router thread accepts requests and routes each one to a thread in a session dedicated to the appropriate transaction management style.

For example, multiple threads in one session might service read-only requests with a database opened for MVCC, while a single thread in another session services update requests with a database opened for update. For information about MVCC and its advantages, see Multiversion Concurrency Control (MVCC) on page 51.

When you are deciding the number of sessions to have in one ObjectStore client process, be sure to weigh the per-session costs, as discussed in Considering Per-Session Costs When Initializing on page 75.

Note that the different transaction management styles need not be handled by sessions that are all in the same ObjectStore client process; it might be beneficial to have multiple application servers processing requests.

Transaction Management Styles

Each style of transaction management varies along the following dimensions:

- *Sharing*: batch or isolated. Transaction management styles vary according to whether they allow processing of multiple requests or a single request per transaction. See Transaction Sharing on page 68.
- *Database access*: MVCC, read-only, or update. Transaction styles vary according to the way the database they operate against is opened, as described in Multiversion Concurrency Control (MVCC) on page 48.
- *Transaction boundaries*, how marked: checkpoint or commit/begin. Transaction boundaries can be marked either with a call to `checkpoint()` or with calls to `commit()` and `begin()`. See Performing Transaction Checkpoints on page 56.

Batch transactions also vary according to the latency between commits or checkpoints, or according to the batch size.

Transaction Sharing

Transaction sharing concerns the number of requests that are serviced in a single transaction. *Isolated* transactions service a single request. *Batch* transactions service multiple requests.

A single transaction can service multiple requests in one of two ways:

- In a single thread by beginning a transaction, servicing several requests, and then committing or checkpointing the transaction.
- In multiple threads of the same session by beginning a global transaction in one thread, servicing several requests in each thread in the session, synchronizing threads, and then committing or checkpointing the transaction in one thread — the same thread that started the transaction.

Using batch transactions can sometimes increase throughput by reducing commit overhead. Use multithreaded batch transactions for operations that cannot conflict with one another — for example, for read-only operations.

It is always safe to use multithreaded batch transactions for read-only operations; but you should be careful when using multithreaded, batch update transactions. If the operations in different threads (in the same session) conflict, the results might be incorrect and transactionally inconsistent data might be recorded in the database.

Batch Transaction Commits and Aborts

If an operation in a batch update transaction aborts, all the operations in the batch must be aborted. Therefore, with batch update transactions, do one of the following:

- Do not reply to a given request until all the requests in the given request's batch have been serviced (that is, until the transaction commits or checkpoints).

- Log requests so that they can be replayed in case of transaction abort.

Results of batched read operations can be returned without waiting for all operations in the batch to commit, because an abort of one (read-only) operation does not affect the results of other operations in the batch.

You should abort a lexical transaction only from the thread that started the transaction with `OS_TXN_BEGIN()`. `err_explicit_abort` is signaled if another thread aborts the transaction.

Overview of Using Sessions

These are basic steps for using the sessions facility:

1 Initialize the sessions facility.

   Call `objectstore::initialize_for_sessions()` instead of `objectstore::initialize()`. This function lets you specify the number of sessions you expect to use. See Initializing the Sessions Facility on page 74.

2 Create sessions.

   Use `os_session::create()` to create a session. A pointer to an instance of `os_session` is returned by each call to `create()`. See Creating and Deleting Sessions on page 76.

3 Set the current session in each thread.

   One way to set the current session is by calling `os_session::set_current()`. You can also sometimes join a given session by dereferencing a pointer associated with the session (sometimes called *thread absorption*). A thread can enter, leave, and change sessions during the course of its execution. A thread can be in at most one session at a time. See Setting and Getting the Current Session on page 76 and Thread Absorption on page 77.

4 Open databases and retrieve data.

   Threads must be within a session when you are opening and closing databases and when you are accessing persistent data.

There are also some optional steps you can take:

- Set default address space partition size after calling `objectstore::initialize_for_sessions()`. See How Partition Size is Determined on page 75.
- Set other session defaults before calling `objectstore::initialize_for_sessions()`. See Setting and Getting Session Defaults and Session-Specific State on page 79.
- Change the size of a session's address space partition before the session has retrieved any persistent data, using the `objectstore::acquire_address_space()` function.

- Set other session-specific state after calling `objectstore::initialize_for_sessions()`. See Setting and Getting Session Defaults and Session-Specific State on page 79.
- Force full initialization of a session. See Initializing an Individual Session on page 75.

Using Pointers Across Sessions

This section describes when you can and cannot use a pointer in more than one session. It discusses three categories of pointers:

- Pointers to transient ObjectStore objects — databases, transient collections, and related transient objects
- Pointers to persistent objects
- Pointers to transient non-ObjectStore objects

Pointers to Transient ObjectStore Objects

A pointer to an instance of any of the following ObjectStore classes can be retrieved only within a session. Each such pointer is associated with the session in which it was retrieved.

- `os_database`
- `os_segment`
- `os_cluster`
- `os_database_root`
- `os_transaction`
- ObjectStore collection classes

You must dereference such a pointer only in its associated session.

`err_wrong_session` is signaled if you dereference such a pointer in a session other than the one in which it was retrieved.

`err_no_session` is signaled if you retrieve or dereference such a pointer when there is no current session.

Example

Following is an example in which a pointer to an `os_database` is retrieved in one session and dereferenced in another:

```
os_session *sess_1 = os_session::create("sess_1");
os_session *sess_2 = os_session::create("sess_2");
os_database *db;
void f()
{
  os_session::set_current(sess_1);
  db = os_database::open("testdb");
  ...
}
```

```
void g()
{
  os_session::set_current(sess_2);
  OS_BEGIN_TXN(tx1, 0, os_transaction::read_only)
    os_database_root *a_root =
      db->find_root("count"); // WRONG -- err_wrong_session
    ...

  OS_END_TXN(tx1)
  ...
  db->close(); // WRONG -- err_wrong_session
}
```

For two sessions to access the same database, each must open the database and retrieve its own associated database pointer:

```
os_session *sess_1 = os_session::create("sess_1");
os_session *sess_2 = os_session::create("sess_2");
os_database *db1;
os_database *db2;
void f()
{
  os_session::set_current(sess_1);
  db1 = os_database::open("testdb");
  ...
  db1->close();
}

void g()
{
  os_session::set_current(sess_2);
  db2 = os_database::open("testdb");

  OS_BEGIN_TXN(tx1, 0, os_transaction::read_only)
    os_database_root *a_root =     db2->find_root("count");
    ...

  OS_END_TXN(tx1)
  ...
  db2->close();
}
```

Pointers to Persistent Objects

A pointer to a persistent object is associated with the session in which the pointer was retrieved from a database. You must not dereference such a pointer in a session other than its associated session.

Caution

If you dereference such a pointer in a session other than the one in which it was retrieved, the error cannot always be detected; either `err_wrong_session` is signaled or unpredictable program failures result.

If you dereference such a pointer when there is no current session, you join the session in which the pointer was retrieved, if the pointer is not currently mapped into virtual memory. If the pointer is already mapped into virtual memory, either `err_no_session` is signaled or unpredictable program failures result. See Setting and Getting the Current Session on page 76.

Example

Following is an example in which a pointer to a persistent object is retrieved in one session and dereferenced in another:

```
os_session *sess_1 = os_session::create("sess_1");
os_session *sess_2 = os_session::create("sess_2");
os_database *db1;
os_database *db2;
int *countp;
void f()
{
  os_session::set_current(sess_1);
  db1 = os_database::open("testdb");
  OS_BEGIN_TXN(tx1, 0, os_transaction::update)
    os_database_root *a_root =       db1->find_root("count");
     countp = (int*) (a_root->get_value());
    ++*countp;
    ...
  OS_END_TXN(tx1)
  ...
}

void g()
{
  os_session::set_current(sess_2);
  db2 = os_database::open("testdb");
  OS_BEGIN_TXN(tx1, 0, os_transaction::update);
  ++*countp; // WRONG -- err_wrong_session, or error not detected
    ...
  OS_END_TXN(tx1)
  ...
}
```

For two sessions to access the same persistent object, each must retrieve its own associated pointer to the object:

```
os_session *sess_1 = os_session::create("sess_1");
os_session *sess_2 = os_session::create("sess_2");
os_database *db1;
os_database *db2;
int *countp1;
int *countp2;
void f()
{
  os_session::set_current(sess_1);
  db1 = os_database::open("testdb");
  OS_BEGIN_TXN(tx1, 0, os_transaction::read_only)
    os_database_root *a_root =       db1->find_root("count");
     countp1 = (int*) (a_root->get_value());
    ++*countp1;
    ...
  OS_END_TXN(tx1)
  ...
}

void g()
{
  os_session::set_current(sess_2);
  db2 = os_database::open("testdb");
  OS_BEGIN_TXN(tx1, 0, os_transaction::read_only);
    os_database_root *a_root =       db2->find_root("count");
```

```
    countp2 = (int*) (a_root->get_value());
   ++*countp2;
    ...
  OS_END_TXN(tx1)
  ...
}
```

Pointers to Transient Non-ObjectStore Objects

Except for pointers to instances of the ObjectStore classes listed in Pointers to Transient ObjectStore Objects on page 70, pointers to transient memory can be retrieved inside or outside any session. In addition, such a pointer can be used in any session.

However, ObjectStore does not automatically synchronize access to transient memory by different threads. It is the user's responsibility to ensure that concurrent access to transient memory does not lead to incorrect results. Following is an example that is missing the proper synchronization:

Example

```
os_session *sess_1 = os_session::create("sess_1");
os_session *sess_2 = os_session::create("sess_2");
int count = 0;
int count *countp = &count;

void f()
{
  os_session::set_current(sess_1);
  ++*countp; // DANGER -- no synchronization
  ...
  os_session::set_current(0);
}
void g() // runs in a different thread from f()
{
  os_session::set_current(sess_2);
  ++*countp; // DANGER -- no synchronization
  ...
  os_session::set_current(0);
}
```

If `f()` and `g()` can run concurrently in different threads, their updates can interfere with one another. The following is one way to provide the proper synchronization, using the UNIX `pthreads` library:

```
os_session *sess_1 = os_session::create("sess_1");
os_session *sess_2 = os_session::create("sess_2");
int count = 0;
int count *countp = &count;
pthread_mutex_t hmCount = PTHREAD_MUTEX_INITIALIZER;

void f()
{
  os_session::set_current(sess_1);
  pthread_lock_mutex(&hmCount);
  ++*count;
  pthread_unlock_mutex(&hmCount)
  ...
  os_session::set_current(0);
}
void g() // runs in a different thread from f()
```

```
{
  os_session::set_current(sess_2);
  pthread_lock_mutex(&hmCount);
  ++*count;
  pthread_unlock_mutex(&hmCount)
  ...
  os_session::set_current(0);
}
```

For concurrent access to persistent objects by different sessions, you do not have to synchronize access; ObjectStore synchronizes the access automatically, just as it does with access by different ObjectStore clients.

Although ObjectStore detects deadlocks resulting from contention for persistent data, undetected deadlocks can result from contention involving a combination of transient and persistent data. Your program is responsible for preventing or detecting such deadlocks.

Cross-Session References

The ObjectStore API provides cross-session references for referencing persistent objects across sessions. The classes `os_reference_cross_session` and `os_Reference_cross_session` implement nonparameterized and templated versions, respectively, of cross-session references.

You can use cross-session references to store a reference to an object in one session, and then read the reference in other sessions, resolving it to obtain either a `void*` pointer or a soft pointer to the referenced object. Cross-session references are transient and therefore cannot be persistently stored.

Write operations to cross-session references are atomic. That is, the user is not required to synchronize two different writes to the same cross-session reference.

For more information about cross-session references, see the following sections of the *C++ API Reference*:

- `os_Reference_cross_session<T>`
- `os_reference_cross_session`

Initializing the Sessions Facility

Applications that use multiple sessions must initialize the sessions facility by calling `objectstore::initialize_for_sessions()` before creating instances of `os_session`. Note that calling `initialize_for_sessions()` replaces the need call to `objectstore::initialize()` that normally must occur in an ObjectStore application.

In fact, calling `objectstore::initialize_for_sessions()` after calling `objectstore::initialize()` signals the `err_misc` exception. However, calling

`objectstore::initialize()` after `objectstore::initialize_for_sessions()` has no effect.

For more information about initializing the sesions facility, see `objectstore::initialize_for_sessions()` in the *C++ API Reference*.

Considering Per-Session Costs When Initializing

The `objectstore::initialize_for_sessions()` function allows you to specify the number of sessions that you expect to create in the current application. When deciding on this number, be sure to consider the cost of a session. Each session has its own cache, address space partition, and commseg. To the extent that an application's session caches contain overlapping data sets, the application makes less efficient use of cache space, address space, and locking resources. Be sure you choose a small enough number of sessions to allow for sufficient resources for each one.

Determining Whether the Sessions Facility Has Been Initialized

You can determine whether the sessions facility has been initialized for multisession use by calling either `objectstore::is_multiple_session_mode()`or `objectstore::is_single_session_mode()`. For information about both functions, see the following section in the *C++ API Reference*:

- `objectstore::is_multiple_session_mode()`
- `objectstore::is_single_session_mode()`

Initializing an Individual Session

Initialization of an individual session (as opposed to initialization of the sessions facility) can be implicit or explicit:

- Implicit initialization occurs when a thread in that session first accesses a database.
- Explicit initialization occurs when the application calls `os_session::force_full_initialization()`; see `os_session::force_full_initialization()` in the *C++ API Reference* for more information.

How Partition Size is Determined

When an individual session is implicitly or explicitly initialized, ObjectStore determines the size of a session's address space partition — that is, the portion of the persistent storage region associated with the session — as follows:

- If `objectstore::acquire_address_space()` has been called during the session, the size specified in the last call to that function is used. You must call this function before the current session is initialized. For more information, see `objectstore::acquire_address_space()` in the *C++ API Reference*..

- Otherwise, if `objectstore::set_default_address_space_partition_size()` has been called subsequent to `objectstore::initialize_for_sessions()`, the size last passed to `set_default_address_space_partition_size()` is used. See `objectstore::set_default_address_space_partition_size()` in the *C++ API Reference*.
- Otherwise, if the environment variable `OS_DEFAULT_AS_PARTITION_SIZE` is set, its value is used; for more information, see `OS_DEFAULT_AS_PARTITION_SIZE` in Chapter 3 of the *Managing ObjectStore*.
- Otherwise, the size is the size of the process-wide persistent storage region divided by the `n_expected_sessions` argument of the first call to `objectstore::initialize_for_sessions()` (rounded up to the nearest 64 KB). See Initializing the Sessions Facility on page 74.

When the session is initialized, `err_address_space_full` is signaled if the session's address space partition is too small or if the partition size is too large to fit in any portion of free space in the process-wide persistent storage region.

Creating and Deleting Sessions

You create a session with the `os_session::create()` function and delete it with the `os_session::operator delete()` function. Calling the `objectstore::shutdown()` function deletes all sessions and their associated transient objects (see Using Pointers Across Sessions on page 70).

For more information about these functions, see the following sections in the *C++ API Reference*:

- `os_session::create()`
- `os_session::operator delete()`
- `objectstore::shutdown()`

Setting and Getting the Current Session

There are two ways to establish a session as the current session for a given thread:

- By calling `os_session::set_current()` in that thread, specifying a pointer to an `os_session` object as the argument. For more information, see `os_session::set_current()` in the *C++ API Reference*.
- By using thread absorption, if the session allows it. If thread absorption is allowed and if there is no current session, dereferencing an unmapped pointer to persistent memory causes the thread to join the session in which the pointer was retrieved. See Thread Absorption on page 77.

To retrieve a pointer to the current session, call `os_session::get_current()`, as described in `os_session::get_current()` in the *C++ API Reference*.

Note that, if there is a current session, it is an error to use pointers to persistent memory that were retrieved in different sessions. ObjectStore signals the `err_wrong_session` exception if you dereference such a pointer and the pointer is not mapped into virtual memory. Whether or not there is a current session, if you dereference a pointer to persistent memory that was retrieved in a different session and the pointer *is* mapped into virtual memory, the error cannot always be detected; either `err_wrong_session` or `err_no_session` is signaled, or unpredictable program failures result.

Mapping pointers into virtual memory

A pointer is not mapped into virtual memory until the first time it is dereferenced in a given process. It is unmapped at the end of each transaction. Each time an unmapped pointer is dereferenced, it is mapped into virtual memory.

Thread Absorption

If a pointer to persistent memory has been retrieved in a given session and is not yet mapped into virtual memory, a thread can join the session by dereferencing the pointer if the thread has no current session. This operation is called *thread absorption*. You can use the `os_session::set_thread_absorption()` function to specify whether or not an individual session allows thread absorption.

If a session does not allow thread absorption, when dereferencing an unmapped pointer while there are multiple threads in the session, only those threads that are in the session will be suspended; threads in other sessions (or outside of all sessions) will continue normal execution.

If a session allows thread absorption (the default setting), when dereferencing an unmapped pointer, all threads in the process, regardless of what session they are in, will be suspended for the duration.

For information about the functions that control thread absorption, see the following sections in the *C++ API Reference*:

- `os_session::set_thread_absorption()`
- `os_session::get_thread_absorption()`

Getting the Session of Pointers to Objects

You can get a pointer to the session in which a pointer to one the following objects was retrieved:

- One of the following transient ObjectStore objects:
  - Databases (`os_database`)
  - Segments (`os_segment`)
  - Clusters (`os_cluster`)
  - Transactions (`os_transaction`)

- Persistent objects

Retrieving the Session for an ObjectStore Object

To get a pointer to the session in which a transient ObjectStore object was retrieved, you use the `session_of()` function, which is defined for the `os_database`, `os_segment`, `os_cluster`, and `os_transaction` classes. You call this function with a pointer to the ObjectStore object as the `this` argument. The function returns a pointer to the session in which the `this` argument was retrieved.

Retrieving the Session for a Persistent Object

To get a pointer to the session in which a pointer to a persistent object was retrieved, use the `os_session::of()` function. This function is useful when working multiple sessions and your application uses pointers to persistent objects across session boundaries. It allows you to get the session in which the pointer was retrieved, establish that session as the current session, and then dereference the pointer. (Dereferencing the pointer in a session other than the one in which the pointer was retrieved results in the err_wrong_session exception; see Pointers to Persistent Objects on page 71.)

The following example shows you how to use `os_session::of()` get the session in which a pointer was retrieved and establish that session as the current session:

```
foo *ptr;
. . .
os_session::set_current(os_session::of(ptr));
```

If the sessions facility is not initialized for multisession use (see Initializing the Sessions Facility on page 74), `os_session::of()` returns a pointer to a session named `Global Session`.

For more information, see the following sections of the *C++ API Reference*:

- `os_cluster::session_of()`
- `os_database::session_of()`
- `os_segment::session_of()`
- `os_transaction::session_of()`
- `os_session::of()`

Using Collections in a Multisession Environment

The collections facility must be used only within a session. All collection APIs access session-specific state, with the following two exceptions:

- To initialize the collections facility, you must call `os_collection::initialize()` within a session. The call initializes the collections facility for the entire process. Calling `os_collection::initialize()` outside any session has no effect.
- To register a rank or hash function, you must call the `os_index_key()` macro within a session. The call registers a rank or hash function for the entire process. Calling `os_index_key()` outside any session has no effect.

Setting and Getting Session Defaults and Session-Specific State

In addition to the collections functions described in Using Collections in a Multisession Environment on page 79, several functions defined by the `objectstore` class can be used for setting and getting either session defaults or session-specific state. These are described in the following sections.

Dual-Purpose Functions

A number of the `objectstore` functions serve a dual purpose, depending on when they are called:

- They set or get session defaults if called before `objectstore::initialize_for_sessions()`.
- They set or get session-specific state if called after `objectstore::initialize_for_sessions()`.

If called before `initialize_for_sessions()`, these dual-purpose functions set or get default values to be used for newly created sessions. If called after `initialize_for_sessions()`, they set or get state for the current session or signal `err_no_session` if there is no current session.

For example, calling `objectstore::set_cache_size()` before calling `objectstore::initialize_for_sessions()` sets the default cache size, that is, the size of the cache assigned to a newly created session. After `initialize_for_sessions()` is called, `set_cache_size()` sets the size for the current session or signals `err_no_session` if there is no current session.

If there are multiple threads before initialization of the sessions facility, you must synchronize their use of these functions. ObjectStore does not provide thread locking before initialization — that is, before a call to `objectstore::initialize()` or `objectstore::initialize_for_sessions()`.

Following are the dual-purpose set functionsof the `objectstore` class:

- `set_allocation_strategy()`
- `set_always_null_illegal_pointers()`
- `set_auto_load_DLLs_function()`
- `set_auto_open_mode()`
- `set_cache_size()`
- `set_current_schema_key()`
- `set_fetch_policy()`
- `set_incremental_schema_installation()`
- `set_suppress_invalid_hard_pointer_errors()`
- `set_lock_option()`
- `set_lock_timeout()`
- `set_null_illegal_pointers()`
- `set_simple_auth_ui()`
- `set_suppress_invalid_hard_pointer_errors()`
- `set_transaction_max_retries()`
- `set_transaction_priority()`

Following are the dual-purpose get functions:

- `get_allocation_strategy()`
- `get_always_null_illegal_pointers()`
- `get_autoload_DLLs_function()`
- `get_auto_open_mode()`
- `get_fetch_policy()`
- `get_incremental_schema_installation()`
- `get_lock_option()`
- `get_lock_timeout()`
- `get_null_illegal_pointers()`
- `get_simple_auth_ui()`
- `get_suppress_invalid_hard_pointer_errors()`
- `get_transaction_max_retries()`
- `get_transaction_priority()`

Single-Purpose Functions

The following members of `objectstore` always set or get process-wide state:

- `embedded_server_available()`
- `get_application_server_pathname()`
- `get_cache_size()`

- get_locator_file()
- get_log_file()
- get_page_size()
- get_thread_locking()
- get_transient_delete_function()
- is_multiple_session_mode()
- is_single_session_mode()
- network_servers_available()
- propagate_log()
- release_maintenance()
- release_major()
- release_minor()
- release_name()
- set_client_name()
- set_default_address_space_partition_size()
- set_thread_locking()
- set_transient_delete_function()
- shutdown()
- which_product()

All other objectstore members access process-specific state or signal err_no_session if there is no current session.

Example

This section discusses an example of a simplified application server that uses two sessions, one for update requests and one for MVCC read requests. The update session uses a single thread while the MVCC session uses multiple threads. The MVCC view is refreshed after a specified latency. The source code for this example is available in $OS_ROOTDIR/examples/sessions.

main()

The main() function, in app_server.cpp, performs the following tasks:

- Initializes the sessions facility
- Creates sessions
- Initializes the application
- Starts up update processing
- Starts up MVCC processing
- Routes requests

- Cleans up update processing
- Cleans up MVCC processing
- Deletes sessions

Most of these tasks are performed by static members of the class `app_server`, which is never instantiated. (A real application server might make these functions nonstatic and instantiate the class, enabling a server to route requests to other servers as well as to a session within its own process space.)

Request Execution

The requests are executed by `app_server::view_seats()` and `app_server::buy_seats()`. In a ticketing application, a request to view the available seats for a specified performance might be handled as an MVCC request; a request to buy specified seats to a specified performance might be handled as an isolated update request.

(A real application server might allow the functions for executing requests to be registered by a separate DLL, along with the associated setup and cleanup functions.)

init()

The function to initialize the application, `app_sever::init()`, creates two queues, one for scheduling requests to be handled by the update session, and the other for scheduling requests to be handled by the MVCC session. It also creates a database with a persistent integer. The update and MVCC operations access this persistent data. (With a real application server, the operations would access more complex data, and the database might already exist.)

Start-up Procedures

The start-up procedures, `app_server::startup_mvcc_processing()` and `app_server::startup_update_processing()`, do the following:

- Set the current session.
- For MVCC start up, record the time just before the database is opened for MVCC. This is used to determine when to refresh the MVCC view with `os_transaction::checkpoint()`.
- Set up the update or MVCC operation (open the database, retrieve the root).
- For MVCC start-up, start a shared transaction. This transaction is committed in `app_server::cleanup_mvcc_processing()`.
- Create worker threads, the threads to which to route requests.
- Unset the current session (that is, set it to `0`).

Each start-up function sets the current session before calling the setup function. Each setup function opens the same database and retrieves a pointer to the same persistent object. However, each opens the database for a different type of access and each has its own associated database pointer and integer pointer.

Thread Functions

The thread functions for the worker threads, `app_server::service_update_requests()` and `app_server::service_mvcc_requests()`, have the following structure:

- Set the current session.
- Repeat: {
- Get a request from `app_server::mvcc_queue` or `app_server::update_queue`.
- For MVCC threads, perform a refresh, if necessary.
- For the update thread, start a transaction.
- Execute the request against the database.
- For the update thread, end the transaction.
- Send the reply.
- } Stop repeating when there is an "end" request.
- Unset the session.

buy_seats() and view_seats()

`buy_seats()` increments a persistent integer, and `view_seats()` simply reads it. Each function replies with the integer's value together with the arguments that were passed to it.

send_reply()

`app_server::send_reply()` simulates sending a reply to an end-user process by sending the values returned by `view_seats()` or `buy_seats()` to standard output. `send_reply()` assumes that exactly three values are always returned, two `floats` and a string. (In a real application server, each operation might have an associated `reply_format` object, which would allow `send_reply()` to be more flexible.)

refresh_if_needed()

The function `app_server::refresh_if_needed()` performs a refresh if the required latency (specified on the command line) has been exceeded. If the latency has been exceeded, this function waits until there is no persistent access, and then refreshes the MVCC view with `os_transaction::checkpoint()`. (When one thread performs `checkpoint()`, there can be no concurrent persistent access by other threads in the same session.) The mutex `hmRefresh` (locked in `service_mvcc_requests()`) provides a barrier so that `refresh_if_needed()` cannot wait forever.

The mutex `hmPersAccess` synchronizes access to the variable `mvcc_persistent_access`. After `refresh_if_needed()` determines that there is no persistent access, it performs `checkpoint()`. Retaining `hmPersAccess` until after `checkpoint()` returns ensures that no MVCC thread initiates persistent access during the refresh. While one thread is in `refresh_if_needed()`, other MVCC threads can decrement `mvcc_refresh_if_needed`, but they can increment it at most once.

get_request()

The function `get_request()` simulates receiving a request from an end-user process by generating requests consisting of a request type (buy, view, or end), an argument list, and a transaction type (MVCC or update — the type of transaction in which the request should be executed). The argument list consists of a count (which `get_request()` increments each time it generates a request) and a string, the name of the request type.

Chapter 4
Data Integrity

The first section in this chapter introduces three data integrity facilities: inverse data members, illegal pointer detection, and protected references. The rest of the chapter covers inverse data members in detail.

Data Integrity Facilities

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 three of the most common integrity maintenance problems:

- Keeping inverse data members synchronized with each other
- Detecting illegal cross-database pointers or pointers to persistent memory
- Detecting references to deleted objects

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 synchronized 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 86.

Illegal Pointers

Another integrity control problem concerns *illegal pointers*. ObjectStore can detect two kinds of illegal pointers:

- Pointers from persistent memory to transient memory
- Illegal cross-database pointers

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 `objectstore::set_null_illegal_pointers()` in the *C++ API Reference*.

References to Deleted Objects

Use an instance of `os_Reference_protected<T>` instead of a pointer when you want ObjectStore to detect references to deleted objects. `os_Reference_protected<T>` has an interface like `os_soft_pointer<T>`. See Using ObjectStore Soft Pointers on page 22.

After the object referred to by an `os_Reference_protected` is deleted, resolution of the `os_Reference_protected` causes an `err_reference_not_found` exception to be signaled. If the referent database has been deleted, `err_database_not_found` is signaled.

See `os_Reference_protected` in the *C++ API Reference*.

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, because 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 can 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.

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. It might do so, for example, so that the end user could do either of the following:

```
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 */
/* relationship* /
somepart->contents.getvalue().insert(otherpart);
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 8 bytes, 4 bytes for the pointer to the `os_collection` and 4 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 by 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 by 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 8 bytes. It mutates to an out-of-line representation automatically 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 the *C++ Collections Guide and 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 and 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

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 the same as those of the corresponding member macro, but without the last argument, as shown 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 works 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 that follow adhere to 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 of interfaces are supported automatically.

See the *C++ Collections Guide and Reference* for descriptions of the 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 that records a note entered by the user. */
class note {

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

    /* Public Data members */
    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() {
  OS_ESTABLISH_FAULT_HANDLER

  /* 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());

  OS_END_FAULT_HANDLER
}
```

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 only applies the coercion operator if it knows that the desired type of the expression is a `node*` pointer. The following does not work correctly:

```
printf("The value of the relationship is %x \n", n1->next );
```

This does not work 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` as the previous one did, but presents to the end user a functional-style interface. This is done exactly as before, except that the relationships themselves are declared private so that the user cannot directly access them by way of the simple data member or relationship styles of 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 does not have to be the case. Even if they were defined by different classes, `node` and `arc` for example, 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() {
OS_ESTABLISH_FAULT_HANDLER

/* 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());

OS_END_FAULT_HANDLER
}
```

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 the *C++ Collections Guide and Reference* for descriptions of the `os_rel_m_m_body()`, `os_rel_m_1_body()`, and `os_relationship_m_m()` macros.

Following 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() {
  OS_ESTABLISH_FAULT_HANDLER

  /* 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");
  }

OS_END_FAULT_HANDLER
}
```

Example: One-to-Many and Many-to-One Relationships

Following 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 *C++ Collections Guide and 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.

```
#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() {
  OS_ESTABLISH_FAULT_HANDLER

  /* 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 */

  OS_ESTABLISH_FAULT_HANDLER
}
```

Following 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 in which the collection involved does not allow duplicates
- Many-to-many relationships in which 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.

The following example shows the way ObjectStore handles one-to-many relationships in which the collection at the many end of the relationship allows duplicates. It also shows the way ObjectStore handles many-to-many relationships in which 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 the number of 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 because 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 because 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 because 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, you can obtain a greater degree of type safety 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:

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 return an `os_Collection<node*>&` rather than an `os_collection&` only.

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 these macros can be found in the *C++ Collections Guide and 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 the *C++ Collections Guide and 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, respectively, are indexable. For nonindexable members, use `os_no_index`. For indexable members, use a call to the macro `os_index()`, indicating the name of the defining class's `os_backptr` member. 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. Following 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 need not be indexable to be used in a path. For an indexable many-valued relationship, specify `os_auto_index`.

Chapter 5
Metaobject Protocol

The information about metaobject protocol (MOP) is organized in the following manner:

Metaobject Protocol (MOP) Overview

The ObjectStore *metaobject protocol* (*MOP*) is a library of classes that allows you to access ObjectStore schema information. Schema information for ObjectStore databases and applications is stored in the form of objects that represent C++ types. These objects are actually instances of ObjectStore *metatypes*, so called because they are types whose instances represent types. Every object representing a type is an instance of a subtype of the metatype `os_type`. For example, objects representing classes (as opposed to built-in types such as `int`) are instances of `os_class_type`, which is derived from `os_type`. (See the hierarchy diagram in The Metatype Hierarchy on page 109).

There are several classes in the metaobject protocol whose instances represent schema objects other than types, such as `os_base_class` and `os_member` and its subtypes. These auxiliary classes are part of the MOP but are not metatypes and are not, therefore, part of the metatype hierarchy. Descriptions of these auxiliary classes begin at `Class os_type` on page 110.

In addition to telling you the way to perform run-time read access to ObjectStore application, compilation, and database schemas, this chapter explains the way to create classes dynamically and add them to ObjectStore database schemas.

MOP Header Files

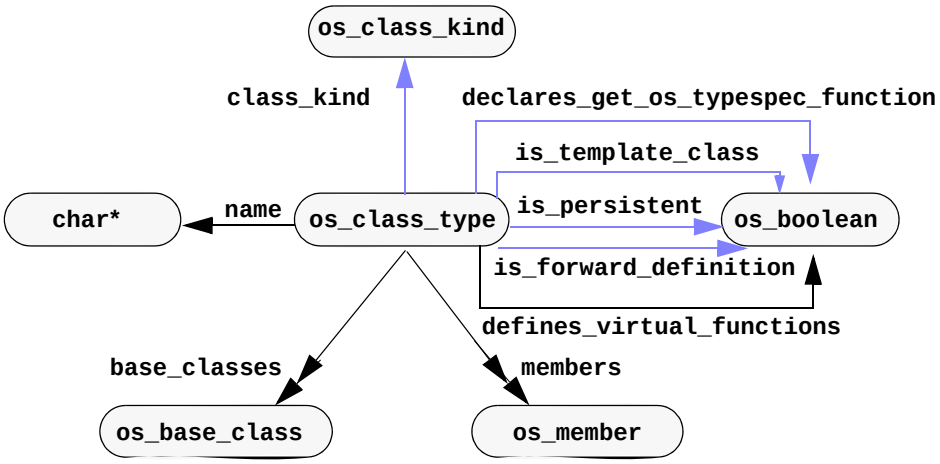
Programs using the metaobject protocol must include `<ostore/ostore.hh>`, followed by `<ostore/coll.hh>` (if a collection is being used), followed by `<ostore/mop.hh>`.

Attributes of MOP Classes

It is useful to think of classes in the MOP as having *attributes*, that is, pieces of abstract state that you can access by using create, get, and set functions.

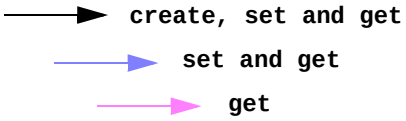
You initialize an attribute with a create function. You perform read access on an attribute with a get function. You update an attribute with a set function. Most of the functions in the MOP are create, get, or set functions, members of the class whose state they access.

Consider, for example, the class `os_class_type`, whose instances represent C++ classes. Following is a diagram showing its attributes:



Each arrow represents an attribute and points to the type of values the attribute has. The arrow's label is the name of the corresponding attribute. The attribute `name`, for example, is string valued. Double arrows indicate a multivalued attribute. For example, the attribute `members` has zero or more `os_members` as its values. These values, for a given instance of `os_class_type`, each represent a member of the class represented by the given `os_class_type`.

Key to arrow shades



The shade of the arrows used throughout this chapter indicates the type of access to the corresponding attribute that is supported. The darkest arrows correspond to attributes with create, get, and set functions. The arrows of medium shade correspond to attributes with get and set functions. The lightest arrows correspond to attributes with get functions only.

create() function for MOP classes

The create function (or functions) for each class is called `create()` and is used to create instances of the class. The get and set functions for each attribute have names based on the attribute name, as follows. For attributes that are not Boolean valued, the functions are

- `get_`*`attribute-name`*`()`
- `set_`*`attribute-name`*`()`

For Boolean-valued attributes, the get and set functions are, respectively,

- *`attribute-name`*`()`
- `set_`*`attribute-name`*`()`

Schema Read Access Compared to Schema Write Access

Many applications that use the MOP perform only read access on schema information. A browser application, for example, might allow viewing of schema information but never perform updates to schemas. Such an application would not use create or set functions.

Schema Read Access

Read access to schema information always starts in one of two ways:

- By looking up a schema object by name in an application, compilation, or database schema
- By retrieving the class of which a specified object is an instance

In either case, a pointer to a `const` object is returned. Other schema objects are the result of performing get functions on these initial `const` objects (and the result of performing get functions on these results, and so on). For class-valued attributes, these get functions, in turn, return pointers or references to `const` objects. Because set functions take only non-`const this` arguments, direct updates to application, compilation, and database schemas are prevented (assuming that you use no explicit casts to non-`const`).

Schema Write Access

To update a database schema, you must first construct, in the transient schema, the classes you want to modify or add to the database schema (see The Transient Schema on page 106). Then you perform installation or evolution on the schema you want to update, specifying the classes in the transient schema as input to the installation or evolution process (see Using MOP for Schema Installation and Evolution on page 108).

Creating a transient schema

Creating classes in the transient schema always starts in one of two ways:

- By invoking a `create()` function
- By copying a class from an application, compilation, or database schema into the transient schema and then looking up the class by name in the transient schema (see The Transient Schema on page 106)

In either case, a reference or pointer to a non-`const` object is returned. The get functions, when performed on non-`const` objects, return non-`const` objects. This is because, except for get functions that return built-in types, get functions come in pairs:

- `const` *`attribute-value-type`* `&get_`*`attribute-name`*`() const ;`
- *`attribute-value-type`* `&get_`*`attribute-name`*`() ;`

or

- `const` *`attribute-value-type`* `*get_`*`attribute-name`*`() const ;`

- *attribute-value-type* *get_*attribute-name*() ;*

Schema Consistency Requirements

Constraints on database schemas

Schema objects in a database schema must meet certain consistency requirements that can be (temporarily) violated by objects in the transient schema. For example, in a database schema, if an `os_class_type`, `c`, has an `os_member`, `m`, as a value of the attribute `members`, then `m` must have `c` as the value for its attribute `defining_class`.

Flexibility of database schemas

To allow flexibility in the construction of schemas, the transient schema has no such restrictions. Therefore, you can, for example, create an `os_member` without first specifying the class that defines it. However, before using classes in the transient schema as input to installation or evolution, you should ensure that the consistency requirements are met. ObjectStore checks such an input for consistency before modifying a database schema and signals `err_mop` if any requirements are not met.

In the MOP interface, pointer arguments to create and set functions generally indicate that `0` is an acceptable initial value for the argument's corresponding attribute; if `0` is not an acceptable value, a *reference* (`&`) argument is used instead of a pointer argument. Note, however, that a nonnull value for the attribute might have to be supplied before installation or evolution to meet the consistency requirements.

For an attribute that must have a nonnull value to meet the consistency requirements, the get functions return a reference type (unless they return a built-in type). When performed on an object that does not yet have a nonnull value for the attribute, such a get function returns an *unspecified* object (see `is_unspecified()` `function` on page 113).

For an attribute that might have a null value and still meet the consistency requirements, the get functions return a pointer type (unless they return a built-in type).

Retrieving an Object Representing the Type of a Given Object

type_at() Function

You can retrieve an instance of `os_type` that represents the type of a given persistent object by passing the object's address to the static member function `os_type::type_at()`. This function is declared as follows:

Declaration

```
static const os_type *type_at(const void *p) ;
```

Note that *p* must point to persistent memory.

For pointers that point to the beginning of more than one object (that is, pointers that point to collocated objects), this function returns the type of the outermost object. For example, consider a pointer typed as a `part*` that points to a direct instance of

mechanical_part, derived from part; and suppose that part has no base types. Passing this pointer to type_at() results in an os_type representing the class mechanical_part, unless the object is embedded as a data-member value in the initial bytes of some other object, in which case the function returns an os_type representing the other object.

Example

Following is an example of the function's use:

```
f () {
part *p = ... ;
. . .
const os_type *t = os_type::type_at(p);
. . .
}
```

type_containing() Function

You can also retrieve the type of the outermost object containing a given persistent object, using os_type::type_containing().

Declaration

```
static const os_type *type_containing(
  const void *p,
  const void*& p_container,
  os_unsigned_int32& element_count
);
```

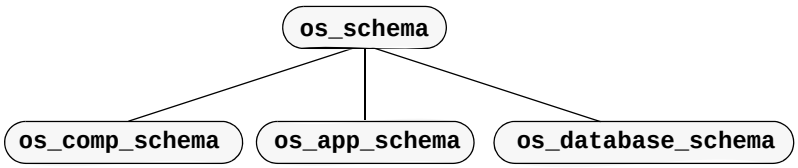
Unlike type_at(), the object whose type is returned does not necessarily begin at the specified address; that is, the argument *p* might point to a subobject or data-member value embedded in the middle of the object whose type is returned.

The address of the object whose type is returned, the outermost object containing the specified object, is referred to by *p_container* when the function returns.

Arrays are handled specially by type_containing(). If the outermost containing object is an array, the element type of the array is returned, and *element_count* is set to refer to the array's size. If the containing object is not an array, *element_count* is set to 0.

Retrieving Objects Representing Classes in a Schema

ObjectStore schemas are represented by instances of classes derived from os_schema.



Instances of these classes represent ObjectStore schemas.

Retrieving database schema

You can retrieve the compilation, application, or database schema in a given database with the static member functions `os_comp_schema::get()`, `os_app_schema::get()`, and `os_database_schema::get()`.

```
static const os_app_schema &os_app_schema::get(
    const database& db);

static const os_comp_schema &os_comp_schema::get(
    const database& db);

static const os_database_schema &os_database_schema::get(
    const database& db);
```

Retrieving application schema

You can also retrieve the application schema for the current application with

```
static const os_app_schema &os_app_schema::get();
```

Retrieving classes in a given schema

You can retrieve objects representing the classes in a given schema with `os_schema::get_classes()`.

```
os_Collection<const os_class_type*>
    os_schema::get_classes() const;
```

You can also retrieve the class with a given name in a given schema with `os_schema::find_type()`.

```
const os_type *os_schema::find_type(
  const char* type_name) const;
```

Retrieving class types

You can retrieve the `os_class_types` in a given *compilation* schema as follows. The examples in this chapter use the parameterized collection classes; if your compiler does not support class templates, use the nonparameterized collection classes instead.

```
f () {
  os_database *my_db =
    os_database::open("/foo/bar/comp_schema");

  os_Collection<const os_class_type*> the_classes =
    os_comp_schema::get(*my_db).get_classes();
. . .
}
```

You can retrieve the `os_class_types` in a given *application* schema in the following way:

```
f () {
  os_database *my_db =
    os_database::open("/foo/bar/app_schema");

  os_Collection<const os_class_type*> the_classes =
    os_app_schema::get(*my_db).get_classes();
. . .
}
```

You can retrieve the types in a *database* schema in the following way:

```
f () {
  os_database *my_db = os_database::open("/foo/bar/db1");
```

```
  os_Collection<const os_class_type*> the_classes =
    os_database_schema::get(*my_db).get_classes();
  . . .
}
```

Finally, you can retrieve the class with a given name in a given schema in the following way:

```
f () {
  os_database *my_db = os_database::open("/foo/bar/db1");

  const os_type *the_type =
    os_database_schema::get(*my_db).find_type("part");
  . . .
}
```

See the entries for the classes `os_schema`, `os_comp_schema`, `os_app_schema`, and `os_database_schema` in Chapter 2 of the *C++ API Reference*.

The Transient Schema

The MOP lets you programmatically update a database's schema. All modification of database schemas, however, is mediated by the transient schema. As described earlier, to update a database schema, you must first construct, in the transient schema, the classes you want to modify or add to the database schema. Then you perform installation or evolution on the database schema, specifying the classes in the transient schema as input to the installation or evolution process.

Initializing the Transient Schema with initialize()

Before using the transient schema, you must always call `os_mop::initialize()`.

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

Recall that creating classes in the transient schema always starts in one of two ways:

- By invoking a `create()` function
- By copying a class from an application, compilation, or database schema into the transient schema and then looking up the class by name in the transient schema

Copying into the Transient Schema with copy_classes()

To copy classes from a schema into the transient schema, you use the static member function `os_mop::copy_classes()`.

```
static void copy_classes(
  const os_schema &schema,
  os_Set<const os_class_type*> &classes
) ;
```

The first argument is the schema containing the classes to be copied and the second argument is a set of pointers to the classes to be copied. Thus, before calling this function, you must perform these steps:

Preparation for the copy operation

1 Retrieve the schema from which you want to copy classes. For example, the following retrieves the application schema for the current application:

```
const os_app_schema &the_app_schema =
  os_app_schema::get() ;
```

2 Before calling `copy_classes()`, you must create a set to hold the pointers to the classes you want to copy:

```
os_Set<const os_class_type*>
  to_be_copied_to_transient_schema ;
```

3 For each class you want to copy, follow these steps:

   a Look up the class to be copied. For example, the following finds the class `os_collection` in the application schema:

```
const os_type *the_const_type_os_collection =
  the_app_schema.find_type("os_collection") ;
```

   b The function `find_type()` can return `0`, so check the result:

```
if (!the_const_type_os_collection)
  error("Could not find the type os_collection in \
      the app schema") ;
```

   c Dereference the result of `find_type()` and convert it from a `const os_type&` to a `const os_class_type&`:

```
const os_class_type *the_const_class_os_collection =
  *the_const_type_os_collection ;
```

   d Insert the class into the set:

```
to_be_copied_to_transient_schema |=
  the_const_class_os_collection ;
```

4 After this is finished for each class you want to copy, you are ready to call `copy_classes()`:

```
os_mop::copy_classes(
  the_app_schema,
  to_be_copied_to_transient_schema
) ;
```

Looking Up a Class in the Transient Schema with find_type()

After you copy a class into the transient schema, you can look it up by name in the transient schema with `os_mop::find_type()`.

```
static os_type *find_type(const char *name) ;
```

Note that `os_mop::find_type()`, unlike `os_schema::find_type()`, returns a pointer to a non-`const os_type`. This means you can modify the `os_type` by performing set functions on it, and you can retrieve other modifiable objects by performing get functions on it. You can also make other objects in the transient schema refer to the `os_type`.

For example, if you do

```
os_type *the_non_const_type_os_collection =
  os_mop::find_type("os_collection") ;
```

you can then create a collection-valued data member (see Class os_member_variable on page 129):

```
assert (the_non_const_type_os_collection) ;

os_member_variable &new_member =
  os_member_variable::create(
    member_name,
    the_non_const_type_os_collection
  ) ;
```

Using MOP for Schema Installation and Evolution

To install classes from the transient schema into a database schema, you use the function `os_database_schema::install()`.

```
void install(os_schema &new_schema) ;
```

The actual argument should be a reference to the transient schema. You can retrieve such a reference with `os_mop::get_transient_schema()`.

```
static os_schema &get_transient_schema() ;
```

The `this` argument is a pointer to the schema you want to modify. Notice that the `install()` function cannot take a `const this` argument. Thus, you cannot use `os_database_schema::get()` to retrieve the schema to be modified. Instead, you use `os_database_schema::get_for_update()`, which takes a `const database&` argument.

```
static os_database_schema &get_for_update(
    const os_database& db)
```

So a call to `install()` might look like

```
os_database_schema::get_for_update(*db).install(
  os_mop::get_transient_schema()
) ;
```

To specify nondefault installation behavior, you can use the alternate overloading of `os_database_schema::install()` that takes an `os_schema_install_options` argument.

```
void install(os_schema &new_schema,
    os_schema_install_options
) ;
```

See `os_schema_install_options` in Chapter 2 of the *C++ API Reference* for a description of the specific installation options available.

The `os_schema_install_options` argument allows you to control whether member functions are copied into the database schema during installation. The default behavior is to not copy member functions.

If you want to modify a database schema in a way that requires evolution, you should use `os_schema_evolution::evolve()` rather than `install()`.

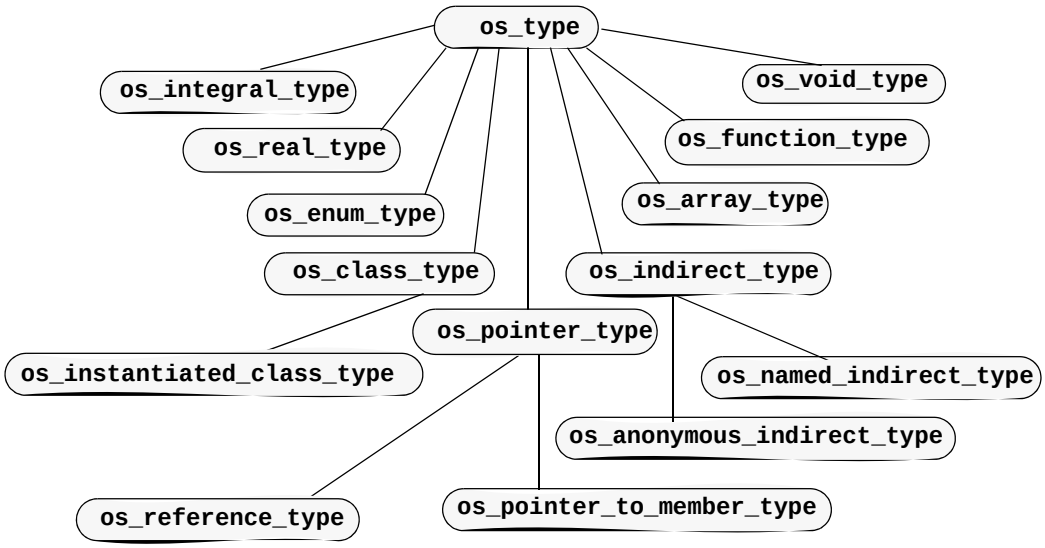
```
static void evolve(
  const char *workdb_name,
  const char *database_name,
  os_schema &new_schema
) ;
```

The *new_schema* argument should be the transient schema. Thus, a call to `evolve()` might look like

```
static void evolve(
  "/example/workdb",
  "/example/partsdb",
  os_mop::get_transient_schema()
) ;
```

The Metatype Hierarchy

All the types in the C++ type system can be divided into the following categories: class types, integer types, real types, enumeration types, array types, pointer types, function types, and the type `void`. For each of these categories, there is a subclass of `os_type` in the metatype hierarchy.



os_instantiated_class_type

The class `os_instantiated_class_type`, derived from `os_class_type`, represents the category of template class instantiations. (The class `os_Collection<part*>`, for example, is an instantiation of the template class `os_Collection`.)

How different types are represented

Pointer types, such as `void*` and `part*`, are represented by direct instances of `os_pointer_type`. Reference types, such as `part&`, are represented by instances of `os_reference_type`, derived from `os_pointer_type`. Pointer-to-member types are represented by instances of `os_pointer_to_member_type`, which is also derived from `os_pointer_type`.

Types with `const` or `volatile` specifiers are represented by instances of `os_anonymous_indirect_type` and `typedefs` are represented by instances of `os_named_indirect_type`.

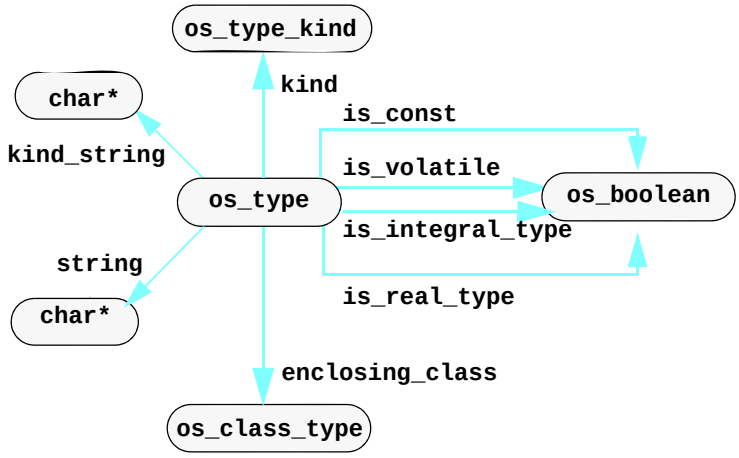
All the classes in the metatype hierarchy are documented in Chapter 2, Class Library, of the *C++ API Reference*. Note that these are not all the types in the metaobject protocol. There are several classes in the metaobject protocol whose instances represent schema objects other than types, such as `os_base_class` (see Class os_base_class on page 123), and `os_member` and its subtypes (see Class os_member on page 126). The rest of this chapter contains a section on each class in the protocol.

Class os_type

The following diagram shows the attributes of the class `os_type`. Each arrow represents an attribute and points to its value type. As described in "Key to arrow shades" on page 101 the darkest arrows have corresponding set functions, get functions, and create arguments. The medium arrows have corresponding set functions and get functions. The lightest arrows have corresponding get functions only.

See `os_type` in Chapter 2 of the *C++ API Reference* for a complete description of this class.

Attributes



create() Function

The `os_type` class defines no `create()` functions. You always create an instance of `os_type` by using a create function defined by one of the subtypes of `os_type`.

kind Attribute

The *kind* of an `os_type` is an enumerator indicating what kind of type is represented by the `os_type`. The enumerators are

kind enumerators

- os_type::Void
- os_type::Named_indirect
- os_type::Anonymous_indirect
- os_type::Char
- os_type::Unsigned_char
- os_type::Signed_char
- os_type::Unsigned_short
- os_type::Signed_short
- os_type::Integer
- os_type::Unsigned_integer
- os_type::Signed_long
- os_type::Unsigned_long
- os_type::Float
- os_type::Double
- os_type::Long_double
- os_type::Pointer
- os_type::Reference
- os_type::Pointer_to_member
- os_type::Array
- os_type::Class
- os_type::Instantiated_class
- os_type::Enum
- os_type::Function
- os_type::Type

Finding the kind of an os_type with get_kind()

The function os_type::get_kind() returns the kind of the specified os_type.

```
os_type_kind get_kind() const ;
```

Given an object typed as an os_type, you can use get_kind() to determine the subtype of os_type that the object is an instance of. Then you can convert the object to that subtype by using the type-safe conversion operators. See Type-Safe Conversion Operators on page 113.

Retrieving the kind_string Attribute

There is a static member function that returns the name of a given os_type_kind:

```
static const char *get_kind_string(os_type_kind) ;
```

For example, os_type::get_kind_string(os_type::Class) returns Class.

Retrieving the string Attribute

The function `os_type::get_string()` returns a new string that contains an expression designating the specified type (such as `part` or `const part`).

```
char *get_string() const ;
```

Note that this function allocates the returned string on the heap, so you should delete it when it is no longer needed.

Determining an os_type's Type and Status

is_const() function

The function `os_type::is_const()` returns nonzero if the specified `os_type` is an `os_anonymous_indirect_type` representing a `const` type (such as `const char*`). Returns `0` otherwise.

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

is_volatile() function

The function `os_type::is_volatile()` returns nonzero if the specified `os_type` is an `os_anonymous_indirect_type` representing a `volatile` type (such as `volatile short`). Returns `0` otherwise.

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

is_integral_type() function

The function `os_type::is_integral_type()` returns nonzero if the specified `os_type` is an instance of `os_integral_type` (such as one representing the type `unsigned int`). Returns `0` otherwise.

```
os_boolean is_integral_type() const ;
```

is_real_type() function

The function `os_type::is_real_type()` returns nonzero if the specified `os_type` is an instance of `os_real_type` (such as one representing the type `long double`). Returns `0` otherwise.

```
os_boolean is_real_type() const ;
```

enclosing_class() function

If a class's definition is nested within that of another class, this other class is the *enclosing class* of the nested class.

There is a pair of `get_enclosing_class()` get functions, one taking `const this` and returning a `const os_class_type*`, and one taking non-`const this` and returning an `os_class_type*`.

- `const os_class_type *get_enclosing_class() const ;`
- `os_class_type *get_enclosing_class() ;`

The function returns `0` if there is no enclosing class.

strip_indirect_types() function

There are also two `os_type::strip_indirect_types()` functions, declared as follows:

- `const os_type &strip_indirect_types() const ;`
- `os_type &strip_indirect_types() ;`

For types with `const` or `volatile` specifiers, this function returns the type being specified as `const` or `volatile`. For example, if the specified `os_type` represents the type `const int`, `strip_indirect_types()` returns an `os_type` representing the

type `int`. If the specified `os_type` represents the type `char const * const`, `strip_indirect_types()` returns an `os_type` representing the type `char const *`.

For `typedefs`, this function returns the original type for which the `typedef` is an alias.

This function calls itself recursively until the result is not an `os_indirect_type`. Consider, for example, an `os_named_indirect_type` representing

```
typedef const part const_part
```

The result of applying `strip_indirect_types()` to this is an `os_class_type` representing the class `part` (not an `os_anonymous_indirect_type` representing `const part`, which would be the result of `os_indirect_type::get_target_type()`).

is_unspecified() function

Some `os_type`-valued attributes in the MOP are required to have values in a consistent schema but might lack values in the transient schema, before schema installation or evolution is performed. The get function for such an attribute returns a reference to an `os_type` or `os_class_type`. The fact that a reference rather than a pointer is returned indicates that the value is required in a consistent schema.

In the transient schema, if such an attribute lacks a value (because you have not yet specified it), the get function returns the unspecified type. This is the only `os_type` for which the following predicate returns nonzero:

```
os_boolean is_unspecified() const ;
```

Type-Safe Conversion Operators

The class `os_type` also defines conversion operators for converting an `os_type&` to a reference to any of the subtypes of `os_type`:

- `operator os_void_type&() ;`
- `operator os_named_indirect_type&() ;`
- `operator os_anonymous_indirect_type&() ;`
- `operator os_integral_type&() ;`
- `operator os_real_type&() ;`
- `operator os_pointer_type&() ;`
- `operator os_reference_type&() ;`
- `operator os_pointer_to_member_type&() ;`
- `operator os_array_type&() ;`
- `operator os_class_type&() ;`
- `operator os_instantiated_class_type&() ;`
- `operator os_enum_type&() ;`
- `operator os_function_type&() ;`

The existence of these operators allows you to supply an expression of type `os_type` or `os_type&` as the actual parameter for a formal parameter of type, for example, `os_integral_type&` — if the designated `os_type` is actually an instance of `os_`

integral_type. If it is not, err_mop_illegal_cast is signaled. Each of these operators is type safe in the sense that err_mop_illegal_cast is always signaled if it is used to perform an inappropriate conversion.

There are also conversion operators for converting a const os_type& to a const reference to any of the subtypes of os_type:

- operator const os_void_type&() const ;
- operator const os_named_indirect_type&() const ;
- operator const os_anonymous_indirect_type&() const ;
- operator const os_integral_type&() const ;
- operator const os_real_type&() const ;
- operator const os_pointer_type&() const ;
- operator const os_reference_type&() const ;
- operator const os_pointer_to_member_type&() const ;
- operator const os_array_type&() const ;
- operator const os_class_type&() const ;
- operator const os_instantiated_class_type&() const ;
- operator const os_enum_type&() const ;
- operator const os_function_type&() const ;

Class os_integral_type

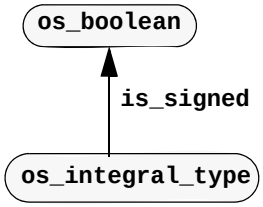
Instances of this class represent one of the following types:

- int
- unsigned int
- short
- unsigned short
- long
- unsigned long
- char
- signed char
- unsigned char

See os_integral_type in Chapter 2 of the *C++ API Reference* for a complete description of this class.

Attribute

This class defines one attribute, `is_signed`:



create() Functions

The `os_integral_type` class has several `create()` functions for the various kinds of integer types:

- `static os_integral_type &create_signed_char() ;`
- `static os_integral_type &create_unsigned_char() ;`
- `static os_integral_type &create_defaulted_char(`
  `os_boolean` *`is_signed`*`) ;`
- `static os_integral_type &create_short(os_boolean` *`is_signed`*`);`
- `static os_integral_type &create_int(os_boolean` *`is_signed`*`) ;`
- `static os_integral_type &create_long(os_boolean` *`is_signed`*`) ;`

Determining a Signed Type with is_signed()

The following function can be used to determine whether the specified type is signed:

```
os_boolean is_signed() const ;
```

Class os_real_type

Instances of this class represent one of the following types:

- `float`
- `double`
- `long double`

See `os_real_type` in Chapter 2 of the *C++ API Reference* for a complete description of this class.

create() Functions

The class `os_real_type` has three `create()` functions, one for each of the real types listed previously.

- `static os_real_type &create_float() ;`
- `static os_real_type &create_double() ;`

- `static os_real_type &create_long_double() ;`

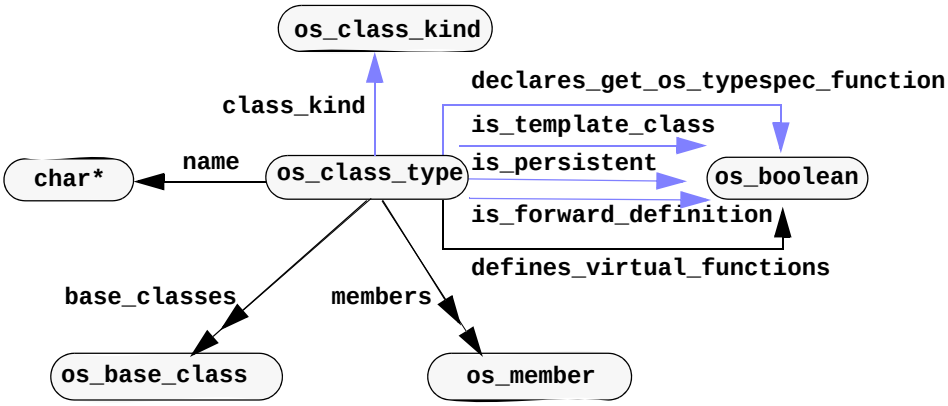
Class os_class_type

An instance of `os_class_type` represents a C++ class. In addition to classes declared with the keyword `class`, `structs` and unions are also represented by instances of `os_class_type`.

See `os_class_type` in Chapter 2 of the *C++ API Reference* for a complete description of this class.

Attributes

The diagram shows some of the important pieces of abstract state associated with the class `os_class_type`. Each arrow represents a piece of state and points to its value type. In the case of double arrows, the value type is an `os_List`, and the type pointed to by the arrow is the element type of the `os_List`.



create() Functions

The class `os_class_type` has two overloadings of the `create()` function. The first overloading is

`static os_class_type &create( const char *name ) ;`

The argument specifies the initial value for the `name` attribute. The initial values for the remaining attributes are as follows:

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

Attribute	*Value*
is_persistent	0
is_forward_definition	1

The second overloading allows specification of more attribute initial values:

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

The arguments specify the initial values for the attributes `name`, `base_classes`, `members`, and `defines_virtual_functions`. The initial values for the remaining attributes are as follows:

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

name Attribute

Getting the attribute

The name of the class represented by a given instance of `os_class_type` is returned by the following function:

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

The returned value points to the beginning of the persistent character array holding the class's name.

Setting the attribute

With the following function you can specify a new character string to serve as a class's name:

```
void set_name(const char* class_type) ;
```

class_kind Attribute

The value of `class_kind` for a given `os_class_type` is an enumerator that indicates the kind of class specified. The enumerators follow:

class_kind enumerators

- `os_class_type::Class`: for a class declared with the keyword `class`
- `os_class_type::Struct`: for a `struct`
- `os_class_type::Union`: for a named union
- `os_class_type::Anonymous_union`: for an anonymous union

Getting the attribute

You can get the `class_kind` with

```
os_unsigned_int32 get_class_kind() const ;
```

Setting the attribute

You can set it with

```
void set_class_kind(os_unsigned_int32) ;
```

members Attribute

With the MOP, the members (both data members and member functions) of a class are sometimes manipulated as a group, using an ObjectStore list of either of the following types:

- `os_List<os_member*>`
- `os_List<const os_member*>`

Each instance of `os_member` represents a member of the class (see Class os_member on page 126). The order of elements in the `os_List` signifies the declaration order of the members.

Getting the attribute

You can retrieve a list of the members of a specified class by using the following functions:

- `os_List<os_member*> get_members() ;`
- `os_List<const os_member*> get_members() const ;`

These functions signal `err_mop_forward_definition` if the value of `is_forward_definition` is nonzero for the specified `os_class_type.`

Setting the attribute

You can specify the members of a class by using

```
void set_members(const os_List<os_member*>&) ;
```

This replaces the members of the specified class with the specified members.

You can also initialize the members of a class by using a `create()` function; see create() Functions on page 116.

os_base_class Objects

As with members, the `os_base_class` objects associated with a class are sometimes manipulated as a group. The list has one of the following two types:

- `os_List<os_base_class*>`
- `os_List<const os_base_class*>`

Each instance of `os_base_class` represents the derivation of one class from another (see Class os_base_class on page 123). The order of elements in the `os_List` signifies the declaration order of the base classes involved.

Retrieving the objects

You can retrieve a list of the `os_base_class` objects for a specified class by using the following functions:

- `os_List<os_base_class*> get_base_classes() ;`
- `os_List<const os_base_class*> get_base_classes() const ;`

These functions signal `err_mop_forward_definition` if the value of `is_forward_definition` is nonzero for the specified `os_class_type`.

Specifying the objects

You can specify the `os_base_class` objects for a class by using

```
void set_base_classes(const os_List<os_base_class*>&) ;
```

This replaces the `os_base_class` objects for the specified class with the specified `os_base_class` objects.

Initialization

You can also initialize the `os_base_class` objects for a class by using a `create()` function; see create() Functions on page 116.

declares_get_os_typespec_function() Function

You can determine whether a given class declares a `get_os_typespec()` member function with

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

which returns nonzero if the class does declare such a member function, and `0` otherwise.

set_declares_get_os_typespec_function() Function

You can specify that a given class declares a `get_os_typespec()` member function with

```
void set_declares_get_os_typespec_function(os_boolean) ;
```

If you supply `1`, the class declares such a member function; if you supply `0`, it does not.

defines_virtual_functions Attribute

The value of this attribute for a given class is nonzero if and only if the class has a field for a pointer to a virtual function table. This value is never computed based on the functions in `members`. It is `0` by default, and nonzero only if so initialized or set by the user.

When you attempt to install a class in a schema, if a function in `members` is virtual, `defines_virtual_functions` must be nonzero, otherwise installation fails.

Getting the attribute

You can retrieve this value with the following function:

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

Setting the attribute

You can set this attribute with

```
void set_defines_virtual_functions(os_boolean) ;
```

Initialization

You also can initialize `defines_virtual_functions` by using a `create()` function; see create() Functions on page 116.

introduces_virtual_functions Attribute

The value of this attribute for a given class is nonzero if the class defines a virtual function but does not inherit any virtual functions.

Getting the attribute

You can retrieve this value with the following function:

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

Setting the attribute

You can set this attribute with

```
void set_introduces_virtual_functions(os_boolean) ;
```

is_forward_definition Attribute

Sometimes a class, `C`, appears in a schema only as a forward definition because some other class uses the type `C*` in its definition (or uses some other pointer type involving `C`) but a full-fledged definition for `C` is not required. For such a class, `is_forward_definition` is nonzero.

Getting the attribute

You can get the value of this attribute with

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

Setting the attribute

You can set this attribute with

```
void set_is_forward_definition(os_boolean) ;
```

is_persistent Attribute

To be installed in a database schema, a class must either be persistent (that is, have a nonzero (true) value for this attribute) or be reachable from a persistent class. Making a class persistent is similar to marking it with `OS_MARK_SCHEMA_TYPE()`.

Getting the attribute

You can get the value of this attribute with

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

Setting the attribute

You can set it with

```
void is_persistent(os_boolean) ;
```

Finding the Nonvirtual Base Class with find_base_class()

The following functions return the `os_base_class` representing the derivation of `this` from the nonvirtual base class with the specified name.

- `const os_base_class *find_base_class(const char *`*`name`*`) const;`
- `os_base_class *find_base_class(const char *`*`name`*`) ;`

The functions return `0` if there is no such base class and signal `err_forward_definition` if `this` is a forward definition.

Finding Base Classes from Which this Inherits with get_allocated_virtual_base_classes()

The `get_allocated_virtual_base_classes()` function returns the following base classes from which `this` inherits:

- `os_List<const os_base_class*>`
- `get_allocated_virtual_base_classes() const ;`
- `os_List<os_base_class*> get_allocated_virtual_base_classes();`

The function returns `0` if there are no such base classes and signals `err_forward_definition` if `this` is a forward definition.

Finding Classes from Which this Indirectly Inherits with get_indirect_virtual_base_classes()

The `get_indirect_virtual_base_classes()` function returns the following base classes from which `this` indirectly and virtually inherits:

- `os_List<const os_base_class*> get_indirect_virtual_base_classes() const ;`
- `os_List<os_base_class*> get_indirect_virtual_base_classes();`

The function returns `0` if there are no such base classes and signals `err_forward_definition` if `this` is a forward definition.

Finding the Member with a Specified Name with find_member_variable()

The `find_member_variable()` function returns the member value of `this` that has the specified name:

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

The function returns `0` if there is no such member.

Finding a Containing Object with get_most_derived_class()

The `get_most_derived_class()` function can be used to determine the object containing a specified data member value and the class of that object.

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

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

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

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

```
<err-0008-0010>Unable to get the most derived class in
os_class_type::get_most_derived_class() containing the
  address 0x[[some-address]].
```

Example

Following is an example:

Class and function declarations

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

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

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

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

Function result

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

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

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

Example

Consider, for example, the following class hierarchy:

Class definitions

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

class B0 {
  public:
```

```
    void f0();
};

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

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

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

Given the following,

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

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

Function result

In this case, `get_most_derived_class()` called on the `pb0` or `pc0` returns a reference to the `os_class_type` for `C0` (the value types of the data member `c0`) and sets *`most_derived_object`* to the same value as *`object`*.

If `B1` is instead defined as

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

and

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

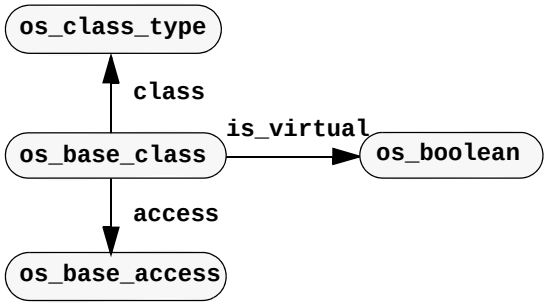
`pb0` and `pi` have the same value because of the optimization, but the base class, `B0`, is collocated with an `int`-valued data member rather than a class-valued data member. `get_most_derived_class()` called on `pb0` or `pi` returns a pointer to the `os_class_type` for the class `C1` and sets *`most_derived_object`*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

Class os_base_class

An instance of `os_base_class` represents the derivation of one class from another. Such an instance serves to associate the base class with the nature of the derivation (virtual or nonvirtual, and public, private, or protected).

See `os_base_class` in Chapter 2 of the *C++ API Reference* for a complete description of this class.

Attributes



create() Functions

The class os_base_class has the following create() function:

```
static os_base_class &create(
  os_unsigned_int32 access_value ,
  os_boolean virtual ,
  os_class_type *associated_class
) ;
```

Function arguments

The arguments specify the initial values for the attributes access, is_virtual, and class.

To represent the derivation of mechanical_part from part, you might create an os_base_class as follows:

Example: os_base_class

```
os_class_type *the_class_part = ...
os_class_type *the_class_mechanical_part = ...
. . .
os_base_class & a_base = os_base_class::create (
  os_base_class::Public ,
  0,
  os_class_type *the_class_part
) ;
os_List<os_base_class*> bases ;
bases |= a_base ;
the_class_mechanical_part->set_base_classes ( bases ) ;
```

class Attribute

The class of an os_base_class is the base class for the derivation represented by the os_base_class. For example, suppose mechanical_part is derived from part. Performing get_base_classes() on an os_class_type representing mechanical_part results in a list containing an os_base_class representing the derivation of mechanical_part from part. Performing get_class() on this os_base_class object results in an os_class_type representing part.

Getting the attribute

You can get the class of a given os_base_class with the following functions:

- const os_class_type &get_class() const ;
- os_class_type &get_class() ;

If no class was specified when the os_base_class was created (that is, if the associated class argument to create() was 0), the unspecified type is returned. This is the only os_type for which os_type::is_unspecified() returns nonzero (see is_unspecified() function on page 113).

Setting the attribute

You can set the class of an os_base_class with

```
void set_class(os_class_type& base_class) ;
```

Initialization

You can initialize the class attribute with os_base_class::create().

access Attribute

The value of the `access` attribute for a given `os_base_class` is one of the following enumerators:

- `os_base_class::Public`
- `os_base_class::Private`
- `os_base_class::Protected`

Getting the attribute

You can get the access for an `os_base_class` with

```
os_unsigned_int_32 get_access() const ;
```

Setting the attribute

You can set the access with

```
void set_access(os_unsigned_int32) ;
```

Initialization

You can initialize the `access` attribute with `os_base_class::create()`.

is_virtual Attribute

The value of `is_virtual` for a given `os_base_class` is nonzero if the `os_base_class` represents a virtual derivation and is `0` otherwise.

Getting the attribute

The following function returns the value of the `is_virtual` attribute:

```
os_boolean is_virtual() const ;
```

Setting the attribute

You can set this attribute with

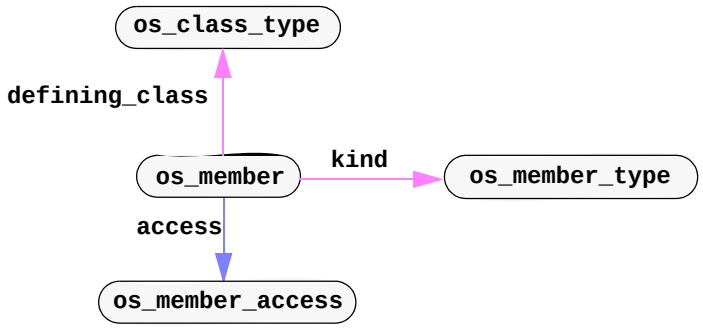
```
void set_is_virtual(os_boolean) ;
```

Class os_member

An instance of `os_member` represents a data member or member function.

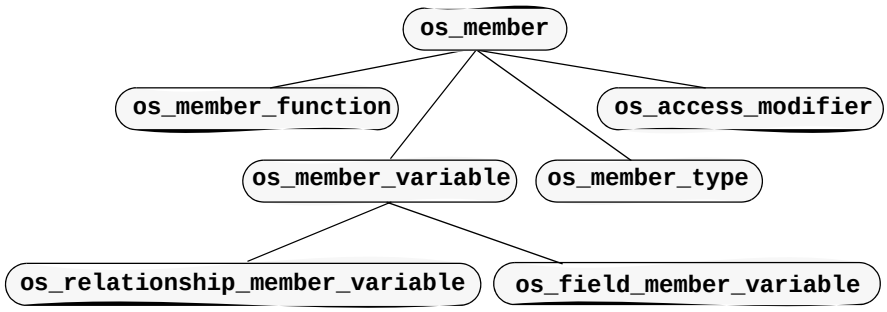
See `os_member` in Chapter 2 of the *C++ API Reference* for a complete description of this class.

Some member functions of os_member



Class os_member and its subclasses

This class has no direct instances. Every instance of `os_member` is a direct instance of one of the subclasses of `os_member`.



create() Functions

Because the class `os_member` has no direct instances, it has no `create()` function. You create an instance of `os_member` with a `create()` function for one of the subclasses of `os_member`.

access Attribute

Attribute enumerators

The `access` of a given `os_member` is one of the following enumerators:

- `os_member::Public`
- `os_member::Private`
- `os_member::Protected`

Getting the attribute

You can get the access of an `os_member` with

```
os_unsigned_int32 get_access() const ;
```

Setting the attribute

You can set the access of an `os_member` with

```
void set_access(os_unsigned_int32 access_mode) ;
```

If the specified *`access_mode`* does not have the same value as one of the preceding enumerators, `err_mop` is signaled.

Initialization

This attribute is initialized to `os_member::Public` by the `create()` functions for the subclasses of `os_member`.

kind Attribute

Attribute enumerators

The `kind` of an `os_member` is one of the following enumerators:

Enumerator	*Use*
`os_member::Variable`	For data members
`os_member::Function`	For member functions
`os_member::Type`	For nested classes

Enumerator	*Use*
`os_member::Access_modifier`	For access declarations
`os_member::Field_variable`	For bit fields
`os_member::Relationship`	For ObjectStore inverse members

Each of these enumerators corresponds to a subclass of `os_member`.

Initialization

The value of `kind` for a given `os_member` is initialized to the enumerator corresponding to the subclass of which the `os_member` is a direct instance. This initialization is performed by the `create()` function for the subclass.

Getting the attribute

You can get the `kind` of an `os_member` with

```
os_unsigned_int32 get_kind() const ;
```

defining_class Attribute

The `defining_class` of an `os_member` is an `os_class_type`, the class that defines the `os_member`.

Getting the attribute

You can get the defining class of an `os_member` with

```
const os_class_type &get_defining_class() const ;
os_class_type &get_defining_class() ;
```

Initialization

This attribute is always initialized by a `create()` function to a special instance of `os_class_type`, the unspecified class (see `is_unspecified() function` on page 113).

Setting the attribute

The `defining_class` of an `os_member` is set automatically when `os_class_type::set_members()` is passed a collection containing a pointer to the `os_member`. The `os_member`'s `defining_class` is set to the `os_class_type` for which `set_members()` is called.

Type-Safe Conversion Operators

Like the class `os_type`, `os_member` defines type-safe conversion operators for converting `const os_members` to `const` references to an instance of a subtype.

- `operator const os_member_variable&() const ;`
- `operator const os_field_member_variable&() const ;`
- `operator const os_relationship_member_variable&() const ;`
- `operator const os_member_function&() const ;`
- `operator const os_access_modifier&() const ;`
- `operator const os_member_type&() const ;`

There are also type-safe conversion operators for converting non-`const` `os_members` to non-`const` references to an instance of a subtype.

- `operator os_member_variable&() ;`
- `operator os_field_member_variable&() ;`

- `operator os_relationship_member_variable&() ;`
- `operator os_member_function&() ;`
- `operator os_access_modifier&() ;`
- `operator os_member_type&() ;`

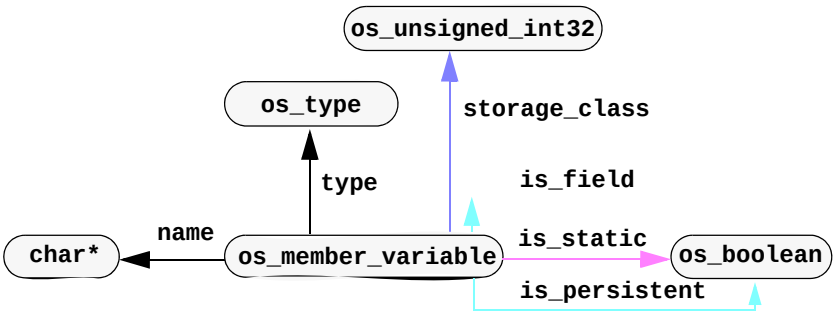
If an attempt is made to perform an inappropriate conversion, `err_mop_illegal_cast` is signaled.

Class os_member_variable

Attributes

The diagram shows some of the important attributes of `os_member_variable`.

See `os_member_variable` in Chapter 2 of the *C++ API Reference* for a complete description of this class.



create() Function

Thes class `os_member_variable` has one `create()` function:

```
static os_member_variable &create(
  const char *unqualified_name ,
  os_type *value_type
) ;
```

Function arguments

The arguments specify the initial values for the attributes `name` and `type`.

The initial values for the remaining attributes are as follows:

Attribute	*Value*
`storage_class`	`os_member_variable::Regular`
`is_field`	`0`
`is_static`	`0`
`is_persistent`	`0`

name Attribute

The name of an os_member_variable is its unqualified name (for example, components rather than part::components).

Getting the attribute

You can get the value of name with

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

Setting the attribute

You can set the value of name with

```
void set_name(const char *name) ;
```

Initialization

You can initialize name with os_member_variable::create().

type Attribute

The type of a given os_member_variable is its value type.

Getting the attribute

You can retrieve an os_member_variable's type with

- `const os_type &get_type() const ;`
- `os_type &get_type() ;`

Setting the attribute

You can set the type with

```
void set_type(os_type& val_type) ;
```

Initialization

You can initialize type with os_member_variable::create().

storage_class Attribute

The value of storage_class for a given os_member_variable is one of the following enumerators:

- os_member_variable::Regular
- os_member_variable::Persistent
- os_member_variable::Static

Initialization

This attribute is initialized to os_member_variable::Regular by os_member_variable::create().

Getting the attribute

You can get the value of storage_class with

```
os_unsigned_int32 get_storage_class() const ;
```

Setting the attribute

You can set storage_class with

```
void set_storage_class(os_unsigned_int32 storage_class) ;
```

is_field Attribute

This attribute is nonzero (true) for all and only instances of os_field_member_variable.

Initialization

It is initialized to nonzero by os_field_member_variable::create(), and is initialized to 0 by os_member_variable::create() and os_relationship_member_variable::create().

Getting the attribute

You can retrieve the value of `is_field` with

```
os_boolean is_field() const ;
```

is_static Attribute

The attribute `is_static` indicates whether a given `os_member_variable` is static.

Initialization

Initializing or setting the attribute `storage_class` causes `is_static` to be set automatically.

Getting the attribute

You can retrieve the value of `is_static` with

```
os_boolean is_static() const ;
```

is_persistent Attribute

The attribute `is_persistent` indicates whether a given `os_member_variable` is a persistent data member.

Initialization

Initializing or setting the attribute `storage_class` causes `is_persistent` to be set automatically.

Getting the attribute

You can retrieve the value of `is_persistent` with

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

Type-Safe Conversion Operators

The class `os_member_variable` defines type-safe conversion operators for converting `const os_member_variables` to `const` references to an instance of a subtype.

- `operator const os_field_member_variable&() const ;`
- `operator const os_relationship_member_variable&() const ;`

There are also type-safe conversion operators for converting non-`const` `os_member_variables` to non-`const` references to an instance of a subtype.

- `operator os_field_member_variable&() ;`
- `operator os_relationship_member_variable&() ;`

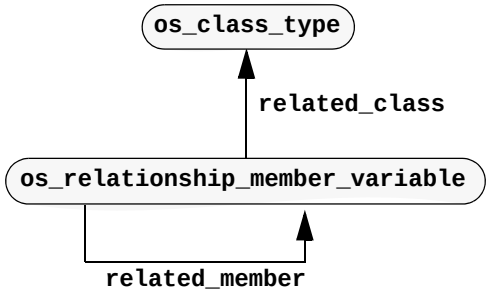
These functions signal `err_mop_illegal_cast` if an attempt is made to perform an inappropriate conversion.

Class os_relationship_member_variable

Attributes

The diagram shows some of the important members of `os_relationship_member_variable`. Attributes inherited from `os_member_variable` and `os_member` are not shown.

See os_relationship_member_variable in Chapter 2 of the *C++ API Reference* for a complete description of this class.



create() Function

The class os_relationship_member_variable has the following create() function:

```
static os_relationship_member_variable &create(
  const char *name,
  const os_type *type,
  const os_class_type *related_class,
  const os_relationship_member_variable *related_member
 );
```

Function arguments

The arguments specify the initial values for the attributes name, type, related_class, and related_member.

The initial values for the remaining attributes are as described for os_member_variable::create().

related_class Attribute

The related_class of a given os_relationship_member_variable is the class defining the inverse of the given member.

Getting the attribute

You can get the related_class with

- const os_class_type &get_related_class() const ;
- os_class_type &get_related_class() ;

Setting the attribute

You can set the related_class with

```
void set_related_class(os_class_type&) ;
```

related_member Attribute

The related_member of a given os_relationship_member_variable is the inverse member of the given member.

Getting the attribute

You can get the value of this attribute with

- const os_relationship_member_variable &
  get_related_member() const ;
- os_relationship_member_variable &get_related_member() ;

Setting the attribute

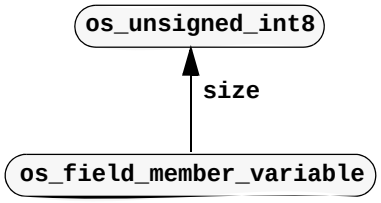
You can set the `related_member` with

```
void set_related_member(os_relationship_member_variable&) ;
```

Class os_field_member_variable

Attribute

The diagram shows the attributes of `os_field_member_variable`. Attributes inherited from `os_member_variable` and `os_member` are not shown.



See `os_field_member_variable` in Chapter 2 of the *C++ API Reference* for a complete description of this class.

create() Function

The class `os_field_member_variable` has the following `create()` function:

```
static os_field_member_variable &create(
  const char *name,
  const os_type *value_type,
  os_unsigned_int8 size_in_bits
) ;
```

Function arguments

The arguments specify the initial values for the attributes `name`, `type`, and `size`. If *`value_type`* is not `0`, a pointer to an `os_integral_type`, or a pointer to an `os_enum_type`, `err_mop` is signaled.

The initial values for the remaining attributes are as described for `os_member_variable::create()`.

size Attribute

The size of a given `os_field_member_variable` is the number of bits in the value of the given member.

Getting the attribute

You can get the `size` with

```
os_unsigned_int8 get_size() const ;
```

Setting the attribute

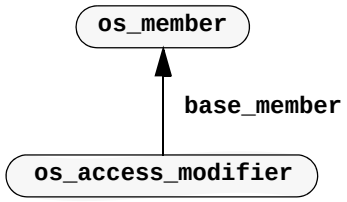
You can set the `size` with

```
void set_size(os_unsigned_int8 size_in_bits) ;
```

Class os_access_modifier

Attribute

An `os_access_modifier` represents an access modification performed by a class on an inherited member. Attributes inherited from `os_member` are not shown.



See `os_access_modifier` in Chapter 2 of the *C++ API Reference* for a complete description of this class.

create() Function

The class `os_access_modifier` has the following `create()` function:

```
static os_access_modifier &create(os_member* mem) ;
```

Function argument

The argument is used to initialize the `base_member` attribute.

base_member Attribute

The `base_member` of a given `os_access_modifier` is the member whose access is being modified.

Getting the attribute

You can get `base_member` with

- `const os_member &get_base_member() const ;`
- `os_member &get_base_member() ;`

Setting the attribute

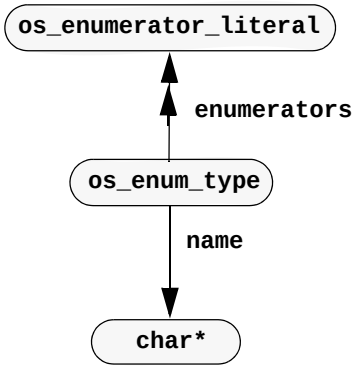
You can set `base_member` with

```
void set_base_member(os_member& mem) ;
```

Class os_enum_type

Attributes

The diagram shows the attributes of `os_enum_type`. Attributes inherited from `os_type` are not shown.



See `os_enum_type` in Chapter 2 of the *C++ API Reference* for a complete description of this class.

create() Function

The class `os_enum_type` has the following `create()` function:

```
static os_enum_type &create(
  const char *name,
  const os_List<os_enumerator_literal*> &enumerators
) ;
```

Function arguments

The arguments initialize the `name` and `enumerators` attributes.

name Attribute

The value of this attribute for an `os_enum_type` is the `os_enum_type`'s name.

Getting the attribute

You can get this attribute with

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

Setting the attribute

You can set the attribute with

```
void set_name(const char *name) ;
```

enumerators Attribute

The value of this attribute for a given `os_enum_type` is a list of the enumerators of the given type, that is, an `os_List<os_enumerator_literal*>`.

Getting the attribute

You can get the value with

- `os_List<const os_enumerator_literal*> get_enumerators() const ;`
- `os_List<os_enumerator_literal*> get_enumerators() ;`

Setting the attribute

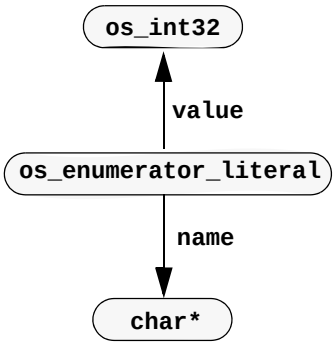
You can set the value with

```
void set_enumerators(const os_List<os_enumerator_literal*>&) ;
```

Class os_enumerator_literal

Attributes

Instances of this class represent enumerators. The following diagram shows the attributes of `os_enumerator_literal`:



See `os_enumerator_literal` in Chapter 2 of the *C++ API Reference* for a complete description of this class.

create() Function

The class `os_enumerator_literal` has the following `create()` function:

```
static os_enumerator_literal &create(
  const char *name,
  os_int32 value
) ;
```

Function arguments

The arguments specify the initial values for the `name` and `value` attributes.

name Attribute

The value of `name` for a given `os_enumerator_literal` is its unqualified name (for example, `ordered` rather than `os_collection::ordered`).

Getting the attribute

You can get `name` with

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

Setting the attribute

You can set `name` with

```
void set_name(const char *name) ;
```

Initialization

The `name` attribute is initialized by the `create()` function.

Class os_void_type

Instances of this class represent the type `void`, which can be used as a return type for functions or can be used in conjunction with `os_pointer_type` to form the type `void*`.

See `os_void_type` in Chapter 2 of the *C++ API Reference* for a complete description of this class.

create() Function

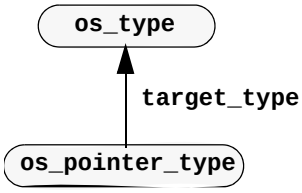
This class `os_void_type` has the following `create()` function is

```
static os_void_type &create() ;
```

Class os_pointer_type

Attribute

Instances of this class are used to represent pointer types.



See `os_pointer_type` in Chapter 2 of the *C++ API Reference* for a complete description of this class.

create() Function

The `create()` function for the class `os_pointer_type` is

```
static os_pointer_type &create(os_type *target_type) ;
```

Function argument

The argument is used to initialize the attribute `target_type`.

target_type Attribute

The value of this attribute for a given `os_pointer_type` is the type of object pointed to by instances of the given pointer type. For example, the `target_type` of an `os_pointer_type` representing `part*` is an `os_type` representing the type `part`.

Getting the attribute

You can get the `target_type` of a given `os_pointer_type` with

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

If no type was specified when the os_pointer_type was created (that is, if the class argument to create() was 0), the unspecified type is returned. This is the only os_type for which os_type::is_unspecified() returns nonzero (see is_unspecified() function on page 113).

Setting the attribute

You can set the target_type with

```
void set_target_type(os_type& target_type) ;
```

This attribute is initialized with the create() function.

Type-Safe Conversion Operators

The class os_pointer_type defines type-safe conversion operators for converting const os_pointer_types to const references to an instance of a subtype.

- operator const os_pointer_to_member_type&() const ;
- operator const os_reference_type&() const ;

There are also type-safe conversion operators for converting non-const os_pointer_types to non-const references to an instance of a subtype.

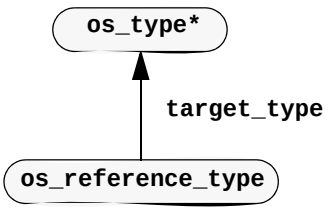
- operator os_pointer_to_member_type&() const ;
- operator os_reference_type&() ;

These functions signal err_mop_illegal_cast if an attempt is made to perform an inappropriate conversion.

Class os_reference_type

Attribute

Instances of this class are used to represent reference types. The diagram shows attributes inherited from os_pointer_type but not those inherited from os_type.



See os_reference_type in Chapter 2 of the *C++ API Reference* for a complete description of this class.

create() Function

The create() function for the class os_reference_type is

```
static os_reference_type &create(os_type *target_type) ;
```

Function argument

The argument is used to initialize the attribute target_type.

target_type Attribute

The value of this attribute for a given `os_reference_type` is the type of object pointed to by instances of the given pointer type. For example, the `target_type` of an `os_reference_type` representing `part&` is an `os_type` representing the type `part`.

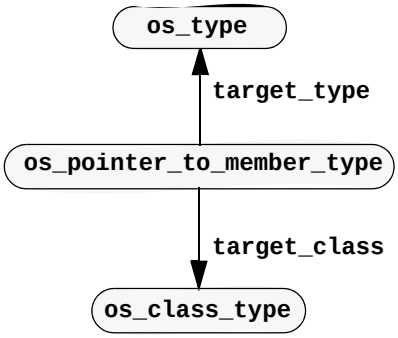
Getting and setting this attribute

You can get and set the `target_type` with functions inherited from `os_pointer_type`.

Class os_pointer_to_member_type

Attributes

Instances of this class are used to represent pointer-to-member types. The diagram shows attributes inherited from `os_pointer_type` but not those inherited from `os_type`.



See `os_pointer_to_member_type` in Chapter 2 of the *C++ API Reference* for a complete description of this class.

create() Function

The `create()` function for the class `os_pointer_to_member_type` is

```
static os_pointer_to_member_type &create(
  os_type *target_type,
  os_class_type *target_class
) ;
```

Function arguments

The arguments are used to initialize the attributes `target_class` and `target_type`.

target_type Attribute

The value of this attribute for a given `os_pointer_to_member_type` is the type of object referred to by instances of the given pointer type.

Getting and setting this attribute

You can get and set the `target_type` with functions inherited from `os_pointer_type`.

target_class Attribute

The value of this attribute for a given `os_pointer_to_member_type` is the class defining the member pointed to.

Getting the attribute

You can get the `target_class` of a given `os_pointer_to_member_type` with

- `const os_class_type &get_target_class() const ;`
- `os_class_type &get_target_class() ;`

If no class was specified when the `os_pointer_to_member_type` was created (that is, if the `target_class` argument to `create()` was `0`), the unspecified type is returned. This is the only `os_type` for which `os_type::is_unspecified()` returns nonzero (see `is_unspecified() function` on page 113).

Setting the attribute

You can set the `target_class` with

`void set_target_class(os_class_type& target_class)`

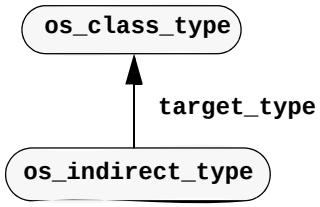
Initialization

This attribute can be initialized with the `create()` function as well.

Class os_indirect_type

Attribute

Instances of this type are direct instances of either `os_indirect_type` or `os_anonymous_indirect_type`.



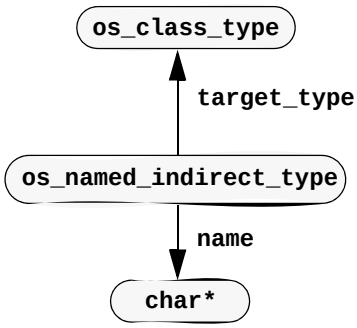
See `os_named_indirect_type` (Class os_named_indirect_type on page 141) and `os_anonymous_indirect_type` (Metaobject Protocol (MOP) Overview on page 100).

See also `os_indirect_type` in Chapter 2 of the *C++ API Reference* for a complete description of this class.

Class os_named_indirect_type

Attributes

An instance of this class represents a C++ `typedef`. The diagram shows the attribute `target_type`, inherited from `os_indirect_type`, but it does not show attributes inherited from `os_type`.



See `os_named_indirect_type` in Chapter 2 of the *C++ API Reference* for a complete description of this class.

create() Function

The `create()` function for the class `os_named_indirect_type` is

```
static os_named_indirect_type &create(
  os_type *target_type,
  const char *name
) ;
```

Function arguments

The arguments are used to initialize `target_type` and `name`.

target_type Attribute

The value of this attribute for a given `os_named_indirect_type` is the type named by the `typedef`.

Getting the attribute

You can get the `target_type` of a given `os_named_indirect_type` with

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

Setting the attribute

You can set the `target_type` with

```
void set_target_type(os_type& target_type) ;
```

Initialization

This attribute can be initialized with the `create()` function.

name Attribute

The value of `name` for a given `os_named_indirect_type` is the name the `typedef` introduces.

Getting the attribute

You can get the name with

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

Setting the attribute

You can set the name with

```
void set_name(const char *name_value) ;
```

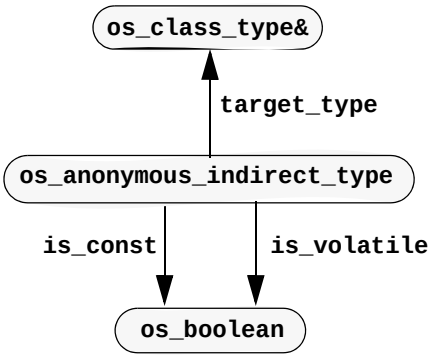
Initialization

name can be initialized by the create() function as well.

Class os_anonymous_indirect_type

Member functions

An instance of this class represents a const or volatile type. The diagram shows the attribute target_type, inherited from os_indirect_type, but it does not show attributes inherited from os_type.



See os_anonymous_indirect_type in Chapter 2 of the *C++ API Reference* for a complete description of this class.

create() Function

The create() function for the class os_anonymous_indirect_type is

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

Function argument

The argument is used to initialize the attribute target_type.

target_type Attribute

The value of this attribute for a given os_anonymous_indirect_type is the type to which the const or volatile specifier applies. For example, the type const int is represented as an instance of os_anonymous_indirect_type whose target_type is an instance of os_integral_type.

Getting the attribute

You can get the target_type of a given os_anonymous_indirect_type with

- `const os_type &get_target_type() const ;`

- `os_type &get_target_type() ;`

Setting the attribute

You can set the `target_type` with

```
void set_target_type(os_type&) ;
```

Initialization

This attribute can be initialized with the `create()` function as well.

is_const Attribute

The value of `is_const` is nonzero for `const` types and `0` otherwise.

Getting the attribute

You can get the value with

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

Setting the attribute

You can set the value with

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

is_volatile Attribute

The value of `is_volatile` is nonzero for volatile types and `0` otherwise.

Getting the attribute

You can get the value with

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

Setting the attribute

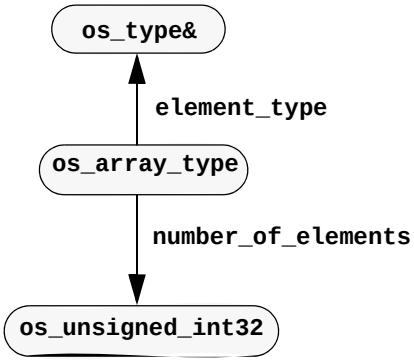
You can set it with

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

Class os_array_type

Attributes

Instances of this class are used to represent array types. Attributes inherited from `os_type` are not shown.



See `os_array_type` in Chapter 2 of the *C++ API Reference* for a complete description of this class.

create() Function

The class `os_array_type` has the following `create()` function:

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

Function arguments

The arguments initialize the attributes `number_of_elements` and `element_type`.

number_of_elements Attribute

The value of this attribute for a given `os_array_type` is the number of elements that instances of the array type are declared to accommodate.

Getting the attribute

You can get this attribute with

```
os_unsigned_int32 get_number_of_elements() const ;
```

Setting the attribute

You can set it with

```
void set_number_of_elements(os_unsigned_int32) ;
```

element_type Attribute

The value of this attribute for a given `os_array_type` is the type of element that instances of the array type are declared to have.

Getting the attribute

You can get this type information with

- `const os_type &get_element_type() const ;`
- `os_type &get_element_type() ;`

Setting the attribute

You can set it with

```
void set_element_type(os_type& elem_type) ;
```

Fetch and Store Functions

ObjectStore provides a number of global (that is, nonmember) functions that allow you to fetch the value of a specified data member (specified with an `os_member_variable`) for a specified object and to store a specified value in a specified data member for a specified object. There are different functions for fetching or storing different types of values.

The first parameter to functions in the `os_fetch` and `os_store classes` is a void pointer to an arbitrary object. This pointer must refer to the closest containing class. If, for example, the relevant member variable being accessed is part of an inherited class, you must pass the address of the base class, not the outermost class. If you pass the wrong pointer, you access the wrong address.

os_fetch() Functions

The `os_fetch()` functions store a reference to the fetched value in the argument `value`, as well as return the value.

For more information, see `::os_fetch()` in Chapter 3 of the *C++ API Reference*.

- `void* os_fetch(`
  `const void *p, const os_member_variable& mem, void *&value);`
- `unsigned long os_fetch(`
  `const void *p, const os_member_variable& mem,`
  `unsigned long &value);`
- `long os_fetch(`
  `const void *p, const os_member_variable& mem, long &value);`
- `unsigned int os_fetch(`
  `const void *p, const os_member_variable& mem,`
  `unsigned int &value);`
- `int os_fetch(`
  `const void *p, const os_member_variable& mem, int &value);`
- `unsigned short os_fetch(`
  `const void *p, const os_member_variable& mem,`
  `unsigned short &value);`
- `short os_fetch(`
  `const void *p, const os_member_variable& mem, short &value);`
- `unsigned char os_fetch(`
  `const void *p, const os_member_variable& mem,`
  `unsigned char &value);`
- `char os_fetch(`
  `const void *p, const os_member_variable& mem, char &value);`
- `float os_fetch(`
  `const void *p, const os_member_variable& mem, float &value);`
- `double os_fetch(`
  `const void *p, const os_member_variable& mem, double &value);`
- `long double os_fetch(`
  `const void *p, const os_member_variable& mem,`
  `long double &value);`

os_store() Functions

For more information, see `::os_store()` in Chapter 3 of the *C++ API Reference*.

- `void os_store(`
  `void *p, const os_member_variable& mem, const void *value);`
- `void os_store(`
  `void *p, const os_member_variable& mem,`
  `const unsigned long value);`

- `void os_store(`
  `void *p, const os_member_variable& mem, const long value);`
- `void os_store(`
  `void *p, const os_member_variable& mem,`
  `const unsigned int value);`
- `void os_store(`
  `void *p, const os_member_variable& mem, const int value);`
- `void os_store(`
  `void *p, const os_member_variable& mem,`
  `const unsigned short value);`
- `void os_store(`
  `void *p, const os_member_variable& mem, const short value);`
- `void os_store(`
  `void *p, const os_member_variable& mem,`
  `const unsigned char value);`
- `void os_store(`
  `void *p, const os_member_variable& mem, const char value);`
- `void os_store(`
  `void *p, const os_member_variable& mem, const float value);`
- `void os_store(`
  `void *p, const os_member_variable& mem, const double value);`
- `void os_store(`
  `void *p, const os_member_variable& mem, const long double value);`

Type Instantiation

You can instantiate the class represented by a given `os_class_type` by calling the global function `::operator new()` without a constructor call. Use the function `os_type::get_size()` to supply the `size_t` argument to `::operator new()`. A `void*` is returned. For example:

```
void *a_part_ptr = ::operator new(
  the_class_part.get_size(),
  db,
  &part_typespec
) ;
```

You can use the function `::os_store()` to initialize the new instance.

Example: Schema Read Access

This section presents an example that illustrates the use of the MOP. The example consists of a routine that prints information about a specified class instance: its class, its address, and the values of its data members. The routine is generic instead of

being class specific, so it can operate on an instance of any class. This is what necessitates the use of the MOP — schema information must be accessed to identify, at run time, the members of the specified object so that their values can be fetched and presented. Something like this functionality might be offered by a browser or debugger.

Top-Level print() Function

The following example involves several functions. The function `print()` is the top-level function. As arguments, it takes a pointer to an object and an `os_class_type&` representing the object's type. There are two other arguments, `member_prefix` and `indentation`, that are supplied only when the function calls itself recursively (see the following example).

print() function definition

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

const os_unsigned_int32 max_buff_size = 100 ;

static void print(const void* p, const os_class_type& c,
  char* member_prefix = "", os_unsigned_int8 indentation = 0
{
  if (!*member_prefix)
    fprintf(stdout, "\n%sclass %s /* 0x%x */ {\n",
      indent(indentation), c.get_name(), p) ;
  os_Cursor<const os_base_class*> bc(c.get_base_classes()) ;
  for (const os_base_class* b=bc.first();b;b = bc.next()) {
    char buff[1024] ;
    os_strcpy(buff, member_prefix) ;
    os_strcat(buff, b->get_class().get_name()) ;
    os_strcat(buff, "::") ;
    print((void*) ((char*)p+b->get_offset()),b->get_class(),
      buff, indentation) ;
  }   /* end of for loop */

  os_Cursor<const os_member*> mc(c.get_members()) ;
  for (const os_member* m=mc.first(); m ; m=mc.next())
      switch (m->kind())
  {
    case os_member::Variable:
    case os_member::Relationship:
    case os_member::Field_variable:
    {
      const os_member_variable& mv = *m ;
      char* type_string = mv.get_type().get_string() ;
      if (mv.is_static() || mv.is_persistent())
        continue ;
      fprintf(stdout, "%s %s\t%s%s = ",
        indent(indentation), type_string,
        member_prefix, mv.get_name()) ;
      print(p, mv, indentation) ;
      fprintf(stdout, " ;\n") ;
      delete [] type_string ;
      break ;
    } /* end of case statement*/
  }   /* end of for loop */

  if (!*member_prefix)
```

```
    fprintf(stdout, "%s}", indent(indentation)) ;
} /* end of print() function */
```

Following is some sample input that explains the function.

Sample input: class definitions

Suppose an application uses the classes `date`, `part`, and `mechanical_part`, with the data members shown in the following definitions:

```
class date {
  private:
    int day;
    int month;
    int year;
  public:
    date(int dd, int mm, int yy) {
      day = dd; month = mm; year = yy;
    }
    . . .
};

class part {
  private:
    int part_id;
    date date_created;
  public:
    part(int id, date d) {part_id = id; date_created = d;}
  . . .
};

class mechanical_part : public part {
  private:
    mechanical_part *parent;
  public:
    mechanical_part(int id, date d, mechanical_part *p) :
      part(id, d) {parent = p;}
  . . .
};
```

Creating objects

And suppose you create objects such as the following:

```
date d(1, 15, 1993);

mechanical_part *parent = new(db) mechanical_part(1, d, 0);

mechanical_part *child = new(db) mechanical_part(2, d, parent);
```

Pass child to print()

Finally, suppose you pass `child` to `print()` as in the following:

```
print(child, os_type::type_at(child));
```

`print()` begins with a check of the argument `member_prefix`, which defaults to a pointer to the null character (`0`). Because this is `0`, you have just started printing an object and the function outputs the name of the object's class with a call to `os_class_type::get_name()`:

```
c.get_name()
```

The object's address is also printed.

Sample output

For the sample input, the output so far might look like the following:

```
class mechanical_part  /* 0xCB320 */ {
```

Recursive Execution of print()

Iterating through the base types

Next, the function iterates through the collection of the specified type's base types, obtained with a call to `os_class_type::get_base_classes()`:

```
c.get_base_classes()
```

For each base class, the function prints the portion of the specified object that corresponds to that base class. It does this by calling itself recursively, specifying the address of the appropriate subobject, and specifying the base type as its type.

How the object's address is obtained

The address of the appropriate object is obtained by adding the base type's offset to `p`, the address of the original object. The offset is obtained by using `os_base_class::get_offset()`.

How the object's type is obtained

The type is obtained by using `os_base_class::get_class()`. Remember, an `os_base_class` encapsulates information about the derivation of one class from another (for example, the offset of the base class within instances of the derived class — which you just used). An `os_base_class` is not itself an `os_class_type`. To get the associated `os_class_type` object, you use `get_class()`.

For the sample input, the only base class is `part`, so an `os_class_type&` representing `part` is passed as the second argument in the recursive call to `print()`.

In addition, because this is a recursive call, a member prefix and indentation are passed as well. The prefix consists of the base type's name followed by `::` (`part::` for the sample input). This is used when printing the names of data members defined by the base type. By using a qualified name for a base class member, the output identifies the defining class.

During the execution of the recursive call, first the member prefix is tested. Because it is nonnull, you do not print the header, `class class-na  /* address */ {`.

Iterating through the base classes of the specified class

Next you iterate through the base classes of the specified class, `part` in this case. This takes care of subobjects corresponding to indirect base classes. Because `part` itself has no base classes, this loop is null for the sample input.

Iterating through members of the specified class

Then you iterate through the collection of members of the specified class, obtained with a call to `os_class_type::get_members()`:

```
c.get_members()
```

Because you are within the recursive execution, the specified class is `part`. You test the kind of each member by using `os_member::kind()`; and for each nonstatic, nonpersistent data member, you output the data member's name (using `member_prefix`) and type, and call an overloading of `print()` that prints data member values (described in the following paragraphs).

Converting os_member to os_member_variable

This involves first converting the `os_member` to an `os_member_variable`:

```
const os_member_variable &mv = *m;
```

Recall that there are type-safe conversions from `os_member` to `const os_member_variable&` and to all the other subtypes of `os_member`.

You get the value type of the member by using `os_member_variable::get_type()` and you get the name of this type by using `os_type::get_string()`:

```
char *type_string = mv.get_type().get_string()
```

Note that `get_string()` allocates a character array, which is deleted when it is no longer needed:

```
delete [] type_string;
```

Sample output: first member of part

For the sample input, the output after retrieving the first member of `part` might look like this:

```
class mechanical_part  /* 0xCB320 */ {
  int part::part_id =
```

print() function for data members

Now consider the function `print()` for data members, which is called in the line

```
print(p, mv, indentation);
```

This function is defined as follows:

```
/* Prints the value at p. It is the value of the data member */
/* indicated by the "m" argument */

static void print(const void* p,
   const os_member_variable& m,
   const os_unsigned_int8 indentation)
{
  const os_type& mt = m.get_type().strip_indirect_types() ;
  if (mt.is_integral_type()) {
    if (((const os_integral_type&)mt).is_signed()) {
      os_int32 value ;
      fprintf(stdout, "%ld", os_fetch(p, m, value)) ;
    } /* end if */

    else {
      os_unsigned_int32 value ;
      fprintf(stdout, "%lu", os_fetch(p, m, value)) ;
    } /* end else */
    return ;
  } /* end if */

  else if (mt.kind()==os_type::Enum) {
    os_int32 value = 0 ;
    const os_enumerator_literal* lit=
      ((os_enum_type&)mt).get_enumerator(
      os_fetch(p, m, value)) ;
    fprintf(stdout, "%ld(%s)", value,
       (lit ? lit->get_name() : "?enum literal?" )) ;
    return ;
  } /* end else if */

switch (mt.kind()) {

  case os_type::Float: {
    float value ;
    fprintf(stdout, "%f", os_fetch(p, m, value)) ;
    return ;
  }
```

```
  case os_type::Double: {
    double value ;
    fprintf(stdout, "%lg", os_fetch(p, m, value)) ;
    return ;
  }

  case os_type::Long_double: {
    long double value ;
    fprintf(stdout, "%lg", os_fetch(p, m, value)) ;
    return ;
  }

  case os_type::Pointer:
  case os_type::Reference:
    print_a_pointer((char*)p+m.get_offset()) ;
    return ;

  case os_type::Class:
  case os_type::Instantiated_class:
    print((char*)p+m.get_offset(),
      (const os_class_type&)mt, "",
      indentation+1) ;
    return ;

  case os_type::Array:
    print((char*)p+m.get_offset(), (const os_array_type&)mt,
      indentation+1) ;
    return ;

  default:
    /* print its address */
    fprintf(stdout, "*0x%x*", (char*)p + m.get_offset()) ;
  } /* end switch */
}
```

Behavior of print() for data members

The `print()` function for data members function begins by retrieving the value type of the specified data member and applying `strip_indirect_types()`. The result is an `os_type` that is not an `os_indirect_type`. Next, it determines the kind of this type and acts accordingly.

For the sample input, the member is currently `part::part_id` and the value type is `int`. This is a signed integer type, so the value is printed with the format "`%ld`". The value is obtained with `os_fetch()`:

```
os_fetch(p, m, value);
```

Sample output: data members

When this function returns, the output is

```
class mechanical_part   /* 0xCB320 */ {
  int part::part_id = 2
```

For the next member, `part::date_created`, the output is supplemented to look like

```
class mechanical_part   /* 0xCB320 */ {
  int part::part_id = 2
  date date_created =
```

before `print()` for member values is called again.

Next recursion of print()

Now the value type of `part::date_created` is determined to be a class, so the original `print()` function is called recursively again. This puts you two levels of recursion down from the top-level execution of `print()`.

```
print((char*)p+m.get_offset(), (const os_class_type&)mt, "",
    indentation+1) ;
```

The arguments are a pointer to the data member value (obtained with the help of `os_member_variable::get_offset()`) and the member's value type (which you cast to a `const os_class_type&`).

This call to `print()` supplements the output with a representation of the date object that serves as the data member value:

```
class mechanical_part /* 0xCB320 */ {
  int part::part_id = 2
  date part::date_created = class date /* 0xCB324 */ {
    int day = 1
    int month = 1
    int year = 1993
  }
```

Exit from base class loop of print()

Now you have finished the portion of the object corresponding to the base class `part`, and you pop up to the top-level `print()` execution and exit from the base class loop. Then you handle the members defined by the object's direct type, `mechanical_part`.

Looping through the class's members

This involves looping through that class's members and presenting the data members. This class defines one data member, `mechanical_part::parent`, whose value type is `part*`. So `print()` supplements the output with the member's name and type name, as follows,

```
class mechanical_part /* 0xCB320 */ {
  int part::part_id = 2
  date part::date_created = class date /* 0xCB324 */ {
    int day = 1
    int month = 1
    int year = 1993
  }
  part* parent =
```

and then calls `print()` for data members.

print_a_pointer() Function

This function determines that the value type of the current member is a pointer type, so it calls `print_a_pointer()` on the data member's address.

print_a_pointer() definition

```
/* print a pointer value along with as much useful */
/* info as possible */

static void print_a_pointer(const void** p)
{
static os_type::os_type_kind string_char =
    char(0x80) > 0 ? os_type::Signed_char
    os_type::Unsigned_char ;

if (*p)
```

```
{
  const os_type* type = os_type::type_at(*(void**)p) ;
  char* tstr = type ? type->get_string() : os_strdup("???") ;
  fprintf(stdout, "(%s*)%#lx%s",
      tstr, (unsigned long)*(void**)p,
      ((type && (type->kind() == string_char)) ?
      get_string((char*)*p, string_char) : "")) ;
  delete tstr ;
  const void* op = 0 ;
  os_unsigned_int32 ecount = 0 ;
  const os_type* otype = os_type::type_containing(
    *p, op, ecount) ;

  if (op && (op != *p) && (otype != type))
  {
    /* the enclosing object is different */

    if (ecount > 1)
    {
      /* point to the appropriate array element */
      os_unsigned_int32 offset = (char*)op - (char*)*p ;
      os_unsigned_int32 i = offset / otype->get_size() ;
      op = (char*)op + (i * otype->get_size()) ;
    } /* end if */

    char* tstr = type ? otype->get_string() : os_strdup("???");
    fprintf(stdout, " /* enclosing object @ (%s*)%#lx */ ",
        tstr, (unsigned long)op) ;
    delete tstr ;
  } /* end if */
} /* end if */

else fprintf(stdout, "0") ;

} /* end print_a_pointer() */
```

`print_a_pointer()` prints the specified pointer's type and value and, if the pointer is a `char*`, it prints up to the first 100 characters of the designated string (with the help of `get_string()`, shown in the following paragraph). In addition, if the object pointed to is embedded in some other object or array, the type and address of the enclosing object are printed.

Sample output from print_a_pointer()

The sample output might look like the following:

```
class mechanical_part /* 0xCB322 */ {
  int part::part_id = 2
  date part::date_created = class date /* 0xCB326 */ {
    int day = 1
    int month = 1
    int year = 1993
  }
  part* parent = (part*) 0xCB300
}
```

Other Data-Handling Routines

Array-valued data members are handled with the following routines.

get_string() function

This function builds a printable representation for a `char*` string and returns it. Only strings up to `max_buff_size` are printed. If they are longer, they are truncated and a trailing `...%d...` is used to indicate the true length.

```
static char* get_string(const char* p, os_type::
    os_type_kind string_char)
{
  const void* op = 0 ; os_unsigned_int32 ecount = 0 ;

  /* Ignore embedded strings for now */
  const os_type* otype =
    os_type::type_containing(p, op, ecount) ;

  /* +5 for the quotes + null character*/
  static char buff[max_buff_size+5];
  char *bp = buff ;
  os_strcpy(bp, " \"") ;
  bp += 2 ;

  if (op && otype && (otype->kind() == string_char)) {
    ecount=ecount-(p-(char*)op);

    /*in case it is pointing into the middle */
    os_unsigned_int32 count =
      (ecount <= max_buff_size)? ecount : max_buff_size ;

    for (; count && (*bp = *p); bp++, p++, count--) ;

    if (*p) {
      /* determine its true length */
      count = ecount - max_buff_size ;
      for (; count && (*p); p++, count--) ;
      ecount -= count ;
      os_sprintf(bp-(3+10+3), "...%d...", ecount) ;
      bp = buff + os_strlen(buff) ;
    } /* end if */

    os_strcpy(bp, "\" ") ;

  } /* end if */

  else buff[0] = 0 ;

  return buff ;

} /* end of get_string() */
```

print() function for an array

This function prints the value at `p` as an array. The array is described by the argument `at`.

```
static void print(const void* p, const os_array_type& at,
    const os_unsigned_int8 indentation)
{
  const os_type& element_type =
        at.get_element_type().strip_indirect_types();
  fprintf(stdout, " { ") ;

  for (int i = 0; i < at.number_of_elements();
      i++, p = (char*)p + element_type.get_size())   {
      print(p, element_type, indentation) ;
```

```
      fprintf(stdout,
      "%s", (i+1) == at.number_of_elements() ?
      " }" : ", ") ;
  } /* end of for loop */

}
```

print() function for a pointer

This function prints the value indicated by the pointer `p`, interpreting it as the type supplied by the argument `et`.

```
static void print(const void* p, const os_type& et,
    const os_unsigned_int8 indentation) {

  switch (et.kind()) {
    case os_type::Unsigned_char:
      fprintf(stdout, "%lu", (os_unsigned_int32)*
        (unsigned char*)p) ;
      break ;

    case os_type::Signed_char:
      fprintf(stdout, "%ld", (os_int32)*(char*)p) ;
      break ;

    case os_type::Unsigned_short:
      fprintf(stdout, "%lu", (os_unsigned_int32)*
        (unsigned short*)p) ;
      break ;

    case os_type::Signed_short:
      fprintf(stdout, "%ld", (os_int32)*(short*)p) ;
      break ;

    case os_type::Integer:
      fprintf(stdout, "%ld", (os_int32)*(int*)p) ;
      break ;

    case os_type::Enum:

    case os_type::Unsigned_integer:
      fprintf(stdout, "%lu", (os_unsigned_int32)*
        (unsigned int*)p) ;
      break ;

    case os_type::Signed_long:
      fprintf(stdout, "%ld", (os_int32)*(int*)p) ;
      break ;

    case os_type::Unsigned_long:
      fprintf(stdout, "%lu", (os_unsigned_int32)*
        (unsigned int*)p) ;
      break ;

    case os_type::Float:
      fprintf(stdout, "%f", *(float*)p) ;
      break ;

    case os_type::Double:
      fprintf(stdout, "%lg", *(double *)p) ;
      break ;
```

```
    case os_type::Long_double:
      fprintf(stdout, "%lg", *(long double *)p) ;
      break ;

    case os_type::Pointer:

    case os_type::Reference:
      print_a_pointer((void**)p) ;
      break ;

    case os_type::Array:
      print(p, (const os_array_type &)et, indentation) ;
      break ;

    case os_type::Class:
    case os_type::Instantiated_class:
      print(p, (const os_class_type&)et, "", indentation+1) ;
      return ;

    default:
      /* a type we do not understand how to print */
      fprintf(stdout, "?%s?", et.kind_string(et.kind())) ;
      break ;

  } /* end of switch */
}
```

indent() function for formatting

This function returns a string of blanks corresponding to the indentation specified by the argument `ilevel`.

```
static const char* indent(os_unsigned_int32 ilevel)
{
  static char indent_string[256] ;
  static os_unsigned_int32 maxilevel = 0, cilevel = 0 ;
  const os_unsigned_int32 indent_tab = 3 ;

  if (ilevel > (256/indent_tab)) ilevel = 256/indent_tab ;

  if (ilevel <= maxilevel) {
    indent_string[cilevel*indent_tab]= ‘ ‘ ;
    indent_string[ilevel*indent_tab]= 0 ;
    cilevel = ilevel ;
    return indent_string ;
  } /* end if */

  os_unsigned_int32 limit = ilevel * indent_tab ;

  for (os_unsigned_int32 i = maxilevel*indent_tab;i<limit; i++)
    indent_string[i] = ‘ ‘ ;
  maxilevel = cilevel = ilevel ;
  return indent_string ;

}
```

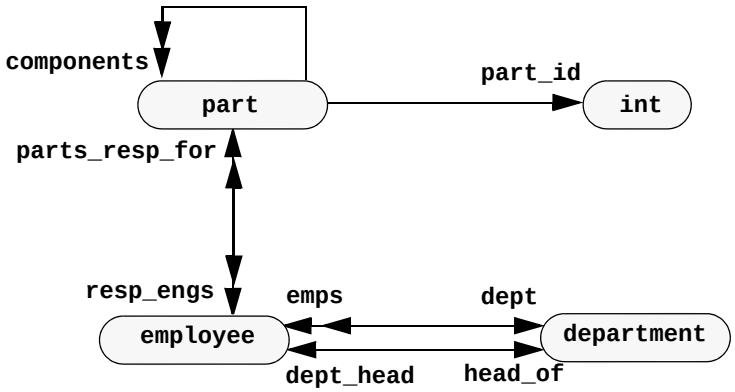
Example: Dynamic Type Creation

Following is an example that uses the MOP to create types and update schemas.

Overview of the gen_schema() Example

The example centers around a function, `gen_schema()`, that might serve as the back end of a much-simplified schema designer application. The front end is a tool for drawing an entity-relationship diagram. An entity-relationship diagram is a graph in which the nodes represent types and the arcs represent possible relationships between instances of the types. The schema designer translates such a diagram into a set of C++ classes, with one or a pair of data members corresponding to each arc in the diagram.

Entity-relationship diagram for the example



The arcs (represented as arrows in the diagram) have single or double arrows at one or both ends. The arrows mean the following:

- Each single or double arrow corresponds to a data member.
- Each single arrow points to the node representing the value type of the corresponding data member.
- Each double arrow corresponds to a collection-valued data member. It points to the node representing the element type of the collection.
- The data member corresponding to a given single or double arrow is defined by the class represented by the node at the other end of the arc containing the arrow.

Schema class definitions for the example

The entity-relationship diagram represents the following schema:

```
class part {
  public:
    int part_id ;
    os_collection &components ;
    os_collection &resp_engs ;
  } ;

class employee {
  public:
```

```
    department *head_of ;
    department *dept ;
    os_collection &part_resp_for ;
  } ;

class department
{
  public:
    employee *dept_head ;
    os_collection &emps ;
} ;
```

If a single arrow points to a node with a class name as label, a pointer to that class is used as the value type of the corresponding data member. This is a simple way to prevent circular dependencies (assuming that arrays of classes are not used). Double arrows correspond to data members whose value type is `os_collection&`. A future release will support the dynamic creation of parameterized types, so it will be possible to use, for example, `os_Collection<part*>&`, instead of `os_collection&`.

gen_schema() Function

Function arguments

The function `gen_schema()` takes as argument an entity-relationship diagram represented as an `os_Collection<arc*>`. An arc has two associated nodes and two associated labels. It also has two associated ends, each of which can have no arrow, a single arrow, or a double arrow.

node and arc class definitions

Following are the definitions of the classes `node` and `arc` as defined in the `graph.hh` header file:

```
/* graph.hh */

#include <string.h>

enum end_enum { no_arrow, single_arrow, double_arrow } ;

class node {
  public:
    char *label ;
    static os_typespec *get_os_typespec() ;
    node ( char *l ) ;
};

class arc {
  public:
    node *node_1 ;
    node *node_2 ;
    end_enum end_1 ;
    end_enum end_2 ;
    char *label_1 ;
    char *label_2 ;

  static os_typespec *get_os_typespec() ;

  arc (
    node *n1,
    node *n2,
    end_enum e1,
    end_enum e2,
```

```
    char *l1,
    char *l2
  ) ;
};
```

node and arc constructors

Following are the implementations of the `node` and `arc` constructors, as defined in the `graph.cc` program file:

```
/* graph.cc */

#include <ostore/ostore.hh>

#include "graph.hh"

arc::arc (
  node *n1,
  node *n2,
  end_enum e1,
  end_enum e2,
  char *l1,
  char *l2
) {
  node_1 = n1;
  node_2 = n2;
  end_1 = e1;
  end_2 = e2;

  if (l1) {
    label_1 = new(
      os_segment::of(this),
      os_typespec::get_char(),
      strlen(l1) + 1
    ) char[strlen(l1) + 1];
  strcpy(label_1, l1);
  } /* end if */
  else
    label_1 = 0;

  if (l2) {
    label_2 = new(
    os_segment::of(this),
    os_typespec::get_char(),
      strlen(l2) + 1
    ) char[strlen(l2) + 1];
  strcpy(label_2, l2);
  } /* end if */

  else
    label_2 = 0;
}

node::node ( char *l ) {
  label = new(
    os_segment::of(this),
    os_typespec::get_char(),
    strlen(l) + 1
  ) char[strlen(l) + 1];
  strcpy(label, l);
}
```

Supporting Functions for the gen_schema() Application

The function `gen_schema()` is supported by five other functions:

- `ensure_in_trans()`
- `copy_to_trans()`
- `add_single_valued_member()`
- `add_many_valued_member()`
- `add_member()`

The function `gen_schema()` processes each `arc` in the diagram one at a time. For each `arc`, it first looks at the two associated nodes. Then `gen_schema()` performs `ensure_in_trans()` on each of the two `node`s.

`ensure_in_trans()` determines whether there is a type in the transient schema whose name is the `node`'s label. If there is not, it determines whether there is a type in the application schema whose name is the `node`'s label. If there is, `ensure_in_trans()` copies it to the transient schema (using `copy_to_trans()`). If there is not, `ensure_in_trans()` creates a class with that name. It returns a pointer to the newly created, copied, or retrieved type.

Copying types from the application schema to the transient schema is the typical means of getting ObjectStore system-supplied classes into the transient schema. However, built-in C++ types, such as `int`, are already present in the transient schema.

Lookups in the application schema and copying from the application schema must be performed within a transaction because they are operations on a database (the application schema database).

Next, `gen_schema()` determines which of the following eight cases applies to the `arc` at hand:

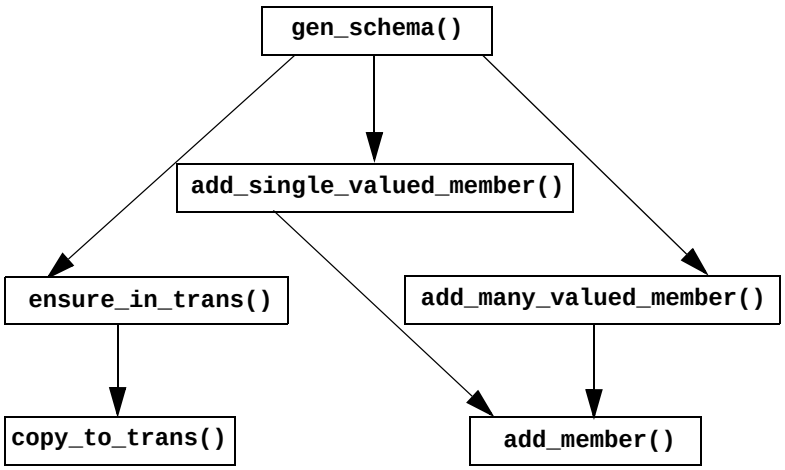
- `end_1` has a single arrow and `end_2` has no arrow.
- `end_1` has no arrow and `end_2` has a single arrow.
- `end_1` has a double arrow and `end_2` has no arrow.
- `end_1` has no arrow and `end_2` has a double arrow.
- `end_1` has a single arrow and `end_2` has a single arrow.
- `end_1` has a double arrow and `end_2` has a single arrow.
- `end_1` has a single arrow and `end_2` has a double arrow.
- `end_1` has a double arrow and `end_2` has a double arrow.

In each case, one or two data members are created, depending on whether there are arrows at one or both ends. A future release will support the dynamic creation of ObjectStore relationship members, so it will be possible to create relationship members in the case in which an `arc` has arrows at both ends. For now, the example creates regular data members.

Each data member is created and added to the appropriate defining class. This is accomplished by `add_single_valued_data_member()` or `add_many_valued_`

member(). Each of these functions creates a data member and then calls add_member(), which adds a specified member to a specified class.

Call Graph of Non-ObjectStore Functions for gen_schema()



After gen_schema() finishes processing all the arcs, the transient schema contains the schema represented by the diagram.

gen_schema.cc Source File

Following is the code for gen_schema() and its supporting functions, all of which is contained in the gen_schema.cc file:

```
/* gen_schema.cc */

#include <ostore/ostore.hh>
#include <ostore/coll.hh>
#include <ostore/mop.hh>
#include <stdlib.h>
#include <iostream>
#include <assert.h>

#include "graph.hh"

void error(char *m) {
  cout << m << "\n" ;
  exit (1) ;
}
```

add_member() function definition

This function makes new_member a member of defining_class.

```
void add_member(os_class_type &defining_class,
  os_member &new_member) {

    os_List<os_member*> members(
      defining_class.get_members()
    );
```

```
  members |= &new_member ;
  defining_class.set_members(members) ;
}
```

`os_class_type::get_members()` returns an `os_List<os_member*>`. To add or remove a member, copy the returned list and update the copy. Then pass the list to `os_class_type::set_members()`.

copy_to_trans() function definition

This function copies the class named `class_name` from the application schema to the transient schema. It returns a pointer to the new copy. If the class cannot be found, the function returns `0`.

```
os_type *copy_to_trans(const char *class_name) {

  OS_BEGIN_TXN(tx1, 0, os_transaction::update)

    const os_type *the_const_type_ptr =
      os_app_schema::get().find_type(class_name) ;
    if (!the_const_type_ptr) return 0 ;

    const os_class_type &the_const_class =
      *the_const_type_ptr ;
    os_Set<const os_class_type*>
      to_be_copied_to_transient_schema ;
    to_be_copied_to_transient_schema |= &the_const_class ;
    os_mop::copy_classes (
      os_app_schema::get(),
      to_be_copied_to_transient_schema
    ) ;

  OS_END_TXN(tx1)
  return os_mop::find_type(class_name) ;
}
```

`os_mop::copy_classes()` requires an `os_Set<const os_class_type*>`. To create this set with the appropriate contents, this function first retrieves a `const os_type*`, dereferences it, and converts it to a `const os_class_type&`. The `const os_class_type&` is then dereferenced and inserted into an `os_Set<const os_class_type*>`. Next, this set is passed to `os_mop::copy_classes()`, which copies the set's element into the transient schema.

Finally, `os_mop::find_type()` is used to retrieve from the transient schema a (non-`const`) `os_type*`, which is returned. The function does not simply return `the_const_type_ptr` because `copy_to_trans()` should return a modifiable object, one to which you can add members. Looking up a class in any schema except the transient schema results in a `const os_type*`. Only a lookup in the transient schema results in a non-`const` `os_type*`.

ensure_in_trans() function definition

If no type named `type_name` is in the transient schema, copy it into the transient schema from the application schema. If no type named `type_name` is in the application schema, create it in the transient schema. The function returns a reference to `type`, named `type_name`, in the transient schema.

```
os_type &ensure_in_trans(const char *type_name){

  os_type *t = os_mop::find_type(type_name) ;
  if (!t)
```

```
    t = copy_to_trans(type_name) ;
  if (!t) {
    os_class_type &c = os_class_type::create(type_name) ;
    c.set_is_forward_definition(0) ;
    c.set_is_persistent(1) ;
    t = &c ;
  } /* end if */

  return *t ;
}
```

When you create a class, the attribute `is_forward_definition` defaults to true. Here it is set to false after creation because `gen_schema()` generates a class definition for each node that represents a class. Similarly, `is_persistent` defaults to false. Here, it is set to true so the new class can be installed in a database schema.

add_single_valued_member() function definition

This function creates an `os_member_variable` with value type `value_type` and makes it a member of `defining_type`. If `member_name` is null, the function prints an error and exits. If `defining_class` is not a class, the exception `err_mop_illegal_cast` is signaled.

```
void add_single_valued_member(
    os_class_type &defining_class,
    os_type &value_type,
    const char *member_name
) {

  if (!member_name)
    error("unspecified member name") ;
  os_member_variable &new_member =
  os_member_variable::create( member_name, &value_type ) ;
  add_member(defining_class, new_member) ;
}
```

While the formal parameter `defining_class` is of type `os_class_type&`, the corresponding actual parameter can be typed as `os_type&`. If you pass in such an actual parameter, the MOP invokes `os_type::operator os_class_type&()`, which converts the actual parameter to an `os_class_type&`. If the object designated by the actual parameter is not really an instance of `os_class_type`, the operator signals `err_mop_illegal_cast`.

add_many_valued_member() function definition

This function creates an `os_member_variable` with value type `os_collection&` and makes it a member of `defining_type`. If `member_name` is null, the function prints an error and exits. If `defining_class` is not a class, `err_mop_illegal_cast` is signaled.

```
void add_many_valued_member(
  os_class_type &defining_class,
  const char *member_name
) {
  if (!member_name)
    error("unspecified member name") ;
  os_type *the_type_os_collection_ptr =
  os_mop::find_type("os_collection") ;
  if (!the_type_os_collection_ptr)
    the_type_os_collection_ptr =
    copy_to_trans("os_collection") ;
```

```
  if (!the_type_os_collection_ptr)
    error("Could not find the class os_collection in the \
    application schema") ;
  os_member_variable &new_member =
  os_member_variable::create(
    member_name,
    &os_reference_type::create(the_type_os_collection_ptr)
  ) ;

  add_member(defining_class, new_member) ;
}
```

This function copies the class `os_collection` from the application schema to the transient schema if it is not already present in the transient schema.

gen_schema() function definition

This function creates classes in the transient schema database based on the graph specified by the `arcs`.

```
void gen_schema(const os_Collection<arc*> &arcs) {

  /*  process each arc in the graph */
  os_Cursor<arc*> c(arcs) ;
  for (arc *a = c.first(); a; a = c.next()) {
    os_type &t1 = ensure_in_trans(a->node_1->label) ;
    os_type &t2 = ensure_in_trans(a->node_2->label) ;

    /*  handle 1 of 8 cases, depending on arc's arrows */

    if ( a->end_1 == no_arrow && a->end_2 == single_arrow )
      if (t2.get_kind() != os_type::Class)
        add_single_valued_member(
        t1, /* defining type */
        t2,/* value type */
        a->label_2 /* member with value type t2 */
      ) ;
      else
        add_single_valued_member(
        t1, /*  defining type */
        os_pointer_type::create(&t2), /*  value type */
        a->label_2 /* of member with value type t2 */
      ) ;

    else if ( a->end_1 == single_arrow && a->end_2 ==
        no_arrow )
      if (t1.get_kind() != os_type::Class)
        add_single_valued_member(
         t2, /*  defining type */
        t1, /*  value type */
        a->label_1 /* member with value type t1 */
      ) ;
      else
        add_single_valued_member(
        t2, /*  defining type */
        os_pointer_type::create(&t1), /*  value type */
        a->label_1 /*member with value type t1 */
      ) ;

    else if ( a->end_1 == no_arrow && a->end_2 ==
        double_arrow )
      add_many_valued_member(
```

```
    t1, /*  defining type */
    a->label_2 /*  name of many-valued member */
  ) ;

else if ( a->end_1 == double_arrow && a->end_2 ==
    no_arrow )
  add_many_valued_member(
    t2, /*  defining type */
    a->label_1 /*  name of many-valued member */
  ) ;

else if ( a->end_1 ==single_arrow && a->end_2 ==
    single_arrow ) {
  /*  binary relationship */
  add_single_valued_member(
    t1, /*  defining type */
    os_pointer_type::create(&t2), /*  value type */
    a->label_2 /* member with value type t2 */
  ) ;
  add_single_valued_member(
     t2, /*  defining type */
     os_pointer_type::create(&t1), /*  value type */
     a->label_1 /* member with value type t1 */
  ) ;
} /* end of else if */

else if ( a->end_1 == single_arrow && a->end_2 ==
    double_arrow ) {
  /*  binary relationship */
  add_single_valued_member(
    t2, /*  defining type */
    os_pointer_type::create(&t1), /*  value type */
     a->label_1 /* member with value type t1 */
  ) ;
  add_many_valued_member(
    t1, /*  defining type */
    a->label_2 /*  name of many-valued member */
  ) ;
} /* end of else if */

else if ( a->end_1 == double_arrow && a->end_2 ==
    single_arrow ) {
  /*  binary relationship */
  add_single_valued_member(
    t1, /*  defining type */
    os_pointer_type::create(&t2), /*  value type */
    a->label_2 /* member with value type t2 */
  ) ;
  add_many_valued_member(
    t2, /*  defining type */
    a->label_1 /*  name of many-valued member */
  ) ;
} /* end of else if */

else if ( a->end_1 == double_arrow && a->end_2 ==
    double_arrow ) {
  /*  binary relationship */
  add_many_valued_member(
    t1, /*  defining type */
    a->label_2 /*  name of many-valued member */
```

```
      ) ;
      add_many_valued_member(
        t2, /*  defining type */
        a->label_1 /*  name of many-valued member */
      ) ;
    } /* end of else if */
  } /*  finish processing arcs (for loop)*/
}
```

Driver Definition

This section describes a driver that creates a graph representing the diagram shown at Call Graph of Non-ObjectStore Functions for gen_schema() on page 161. It then passes the graph to `gen_schema()`, which updates the transient schema. Next, the driver installs in the schema of a specified database those classes that are in the transient schema. Finally, it creates an instance of each dynamically created class.

The driver relies on two functions, `find_class()` and `find_member_variable()`, to instantiate the dynamically created classes. These supporting functions are shown first.

find_class() function definition

This function returns a reference to the class in `the_schema` named `class_name`. If the class is not found, the function returns an error. If `class_name` is not a class, `err_mop_illegal_cast` is signaled.

```
const os_class_type &find_class(
  const char *class_name,
  const os_schema &the_schema
) {

  const os_type *the_type_ptr =
    the_schema.find_type(class_name) ;
  if (!the_type_ptr)
    error("Cannot find class with specified name") ;
  return *the_type_ptr ;
}
```

find_member_variable() function definition

This function returns a reference to the member of `defining_class` named `member_name`. If `member_name` is not found, the function returns an error. If `member_name` is not a data member, `err_mop_illegal_cast` is signaled.

```
const os_member_variable &find_member_variable(
  const os_class_type &defining_class,
  const char *member_name
) {

  const os_member *the_member =
    defining_class.find_member_variable(member_name);
  if (!the_member)
    error("Could not find member with specified name.");
  return *the_member ;
}
```

Driver main() function definition

The following code is the `main()` function for the dynamic type creation example.

```
void main(int, char **argv) {

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

```
  os_mop::initialize() ;

  if (!argv[1])
    error("null database name\n") ;

/* create a graph representing an entity-relationship diagram */
/* the graph is a collection of arcs */

  os_Collection<arc*> &arcs =
  os_Collection<arc*>::create(
    os_database::get_transient_database()
  ) ;

  node *part_node = new node("part") ;
  node *employee_node = new node("employee") ;
  node *int_node = new node("int") ;
  node *department_node = new node("department") ;
  arcs |= new arc(
    part_node,
    int_node,
    no_arrow,
    single_arrow,
    0,
    "part_id"
  ) ;

  arcs |= new arc(
    part_node,
    part_node,
    no_arrow,
    double_arrow,
    0,
    "components"
  ) ;

  arcs |= new arc(
    employee_node,
    department_node,
    single_arrow,
    single_arrow,
    "dept_head",
    "head_of"
  ) ;

  arcs |= new arc(
    employee_node,
    department_node,
    double_arrow,
    single_arrow,
    "emps",
    "dept"
  ) ;

  arcs |= new arc(
    part_node,
    employee_node,
    double_arrow,
    double_arrow,
    "parts_resp_for",
    "resp_engs"
```

```
  ) ;

  cout << "Calling gen_schema() ...\n" ;
  gen_schema(arcs) ;
  cout << "Schema generated. Installing schema ...\n" ;
  os_database *db = os_database::open(argv[1], 0, 0664) ;

/*  install schema in db */

  OS_BEGIN_TXN(tx1, 0, os_transaction::update)
    os_database_schema::get_for_update(*db).install(
      os_mop::get_transient_schema()
    ) ;
  OS_END_TXN(tx1)

  OS_BEGIN_TXN(tx2, 0, os_transaction::update)
    /*  create a part, an employee, and a department */
    /*  and partially initialize them */

    os_typespec part_typespec("part") ;
    os_typespec employee_typespec("employee") ;
    os_typespec department_typespec("department") ;

    const os_database_schema &the_database_schema =
      os_database_schema::get(*db) ;
    const os_class_type &the_class_part = find_class( "part",
      the_database_schema ) ;
    const os_class_type &the_class_employee = find_class(
        "employee", the_database_schema ) ;

    void *a_part_ptr = ::operator new(
      the_class_part.get_size(),
      db,
      &part_typespec
    ) ;

    void *an_emp_ptr = ::operator new(
      the_class_employee.get_size(),
      db,
      &employee_typespec
    ) ;

    void *a_dept_ptr = ::operator new(
      find_class( "department",
        the_database_schema ).get_size()
        db,
        &department_typespec
    ) ;

    os_collection &the_components_coll =
    os_collection::create( os_segment::of(a_part_ptr) ) ;
    os_collection &the_resp_engs_coll =
      os_collection::create( os_segment::of(a_part_ptr) ) ;

    the_resp_engs_coll |= an_emp_ptr ;

    os_store(
      a_part_ptr,
      find_member_variable(the_class_part, "part_id"),
      1
```

```
    ) ;

    os_store(
      a_part_ptr,
      find_member_variable(the_class_part,
        "resp_engs"),
      &the_resp_engs_coll
    ) ;

    os_store(
      a_part_ptr,
      find_member_variable(the_class_part,
        "components"),
      &the_components_coll
    ) ;

    os_store(
      an_emp_ptr,
      find_member_variable(the_class_employee,
        "dept"),
      a_dept_ptr
    ) ;

    db->create_root("part_root")->set_value(
        a_part_ptr, &part_typespec ) ;

  OS_END_TXN(tx2)
  db->close() ;
  cout << "Done.\n" ;
}
```

How the driver works

The driver performs schema installation by using `os_database_schema::install()`. To perform installation on a database schema, you must retrieve the schema with `os_database_schema::get_for_update()` instead of `os_database_schema::get()`. The function `os_database_schema::get()` returns a `const os_database_schema&` and `install()` requires a non-`const schema&`.

The driver instantiates the classes `part`, `employee`, and `department` by calling the global function `::operator new()` without a constructor call. The function `os_type::get_size()` is used to supply the `size_t` argument to `::operator new()`. The function `::os_store()` is used to partially initialize the instance of `part`.

You can run this program and use the browser to verify that it produces the class definitions presented at “Schema class definitions for the example” on page 157.

Chapter 6
Dump/Load Facility

The ObjectStore dump/load facility allows you to

- Dump to an ASCII file the contents of a database or group of databases
- Generate a loader executable capable of creating, given the ASCII as input, an equivalent database or group of databases

The dumped ASCII has a compact, human-readable format. You use the ASCII as input to the loader.

By default, objects are dumped in terms of the primitive values they directly or indirectly contain. You can use the default dump and load processes or customize the dumping and loading of particular types of objects. You can, for example, dump and load objects in terms of sequences of high-level API operations needed to recreate them, rather than in terms of the primitive values they contain. This is appropriate for certain location-dependent structures, such as hash tables.

To enhance efficiency during a dump, database traversal is performed in address order whenever possible. To enhance efficiency during loads, loaders are generated by the dumper and tailored to the schema involved. This eliminates most run-time schema lookups during the loading.

Read about the dump/load facility in osdump in Chapter 4 of *Managing ObjectStore* before you read the discussion in this chapter. This chapter addresses the following topics:

When Is Customization Required?

In most cases, customization is unnecessary. The basic types, pointers, ObjectStore references, collections, indexes, and most instances of classes are handled without any customization.

However, you might want to use customization to

- Change representation
- Improve locality
- Reduce the size of the dump output file
- Make the dump format more readable

There are also some circumstances when you *must* take advantage of specialization. You must customize the dumping and loading of C++ unions. In addition, you might have to customize the dumping and loading of objects whose structure depends on the locations of other objects.

A dumped object and its equivalent loaded object do not necessarily have the same location, that is, the same offsets in their segment. Among the implications of this are the following:

- Other objects might use different pseudoaddresses (the identifier a segment uses for an object pointed to by that segment) to refer to the dumped and loaded objects.
- Their addresses might hash to different values; that is, for example, `objectstore::get_pointer_numbers()` might return different values for the dumped and loaded objects.

The default dumper and loader take into account the first implication, and the loader automatically adjusts all pointers in loaded databases to use the new locations. The default dumper and loader also take into account the second implication for ObjectStore collections with hash-table representations. Because a dumped collection element hashes to a different value than the corresponding loaded element does, the dumped and loaded element's hash-table slots are different. The facility, then, does not simply dump and load the array of slots based on fundamental values (which would result in using the same slot for the dumped and loaded objects).

Instead, it dumps the collection in terms of sequences of high-level API operations (that is, string representations of `create()` and `insert()` arguments) that the loader can use to recreate the collection with the appropriate membership.

The default dumper and loader do not take into account the second implication for non-ObjectStore classes. If you have collection classes that use hash-table representations, you must customize their dumping and loading. Any other location-dependent details of data structures (such as encoded offsets) should also be dealt with through customization.

Although the facility provides a great deal of flexibility, customization typically takes the same form as the ObjectStore collection customization previously described.

Customizing Dumps

If you want to dump and load a database containing a location-dependent data structure, you should dump and load the data structure in terms of a sequence of operations that recreates the data structure. The dumper emits the arguments for each operation in the sequence, and the loader recreates the data structure by performing the operations using the arguments in the dumped ASCII.

Creation Stages

Typically, this sequence of operations can be divided into two stages, corresponding to two stages of loading a data structure:

- Initialization stage: This stage creates an instance of the structure in a location-independent state. For example, it creates an empty collection. The object created is called the *root* of the dumped object.
- Fixup stage: This stage performs operations on the root portion to recreate the dumped object in the appropriate location-dependent state. For example, this stage inserts elements into the empty collection. Any additional objects created as a result of these fixup operations are called *nonroot* objects.

Because some objects required for the fixup stage might not exist during the initialization stage, the loader must typically perform the initialization stage, load other objects, then perform the fixup stage. This means that the dumper must dump the arguments for the initialization stage, dump all other objects (except nonroot objects), then dump the arguments for the fixup stage.

For example, when the root of a collection is loaded, the collection elements might not yet exist, so the loader usually creates an empty collection, loads the elements (as well as other objects), then inserts the elements. And the dumper usually dumps the `create()` arguments, dumps the elements (and all other objects except nonroot objects), then dumps the `insert()` arguments.

It is important to distinguish between nonroots of a data structure and objects that are not part of the data structure at all. For example, if a collection's elements are pointers, the pointer objects are nonroots of the collection's data structure, but the objects pointed to by the elements are not part of the data structure at all.

Dumper Actions

To accommodate these stages, the dumper operates in three different modes, performing different kinds of actions in each mode:

- Plan mode: While in plan mode, the dumper invokes type-specific planner actions to identify the nonroot portions of dumped objects. Planner actions store this information, which the dumper accesses when in object-dump mode, to avoid dumping nonroot portions. Because nonroot portions of objects are effectively dumped in fixup-dump mode, they must be ignored while in object-dump mode.
- Object-dump (object form generation) mode: While in object-dump mode, the dumper invokes type-specific object-dumper actions, which typically emit strings

from which the loader can reconstruct arguments. The loader creates the root object by passing the arguments to a high-level API (such as a `create()` function). Object dumpers also create *fixup dumpers*.

- Fixup-dump (fixup form generation) mode: While in fixup-dump mode, the dumper invokes type-specific fixup-dump actions, which typically emit strings from which the loader can reconstruct arguments. The loader updates the root portion and creates the nonroot portion by passing the arguments to a high-level API (such as an `insert()` function).

The following pseudocode summarizes the flow of control among modes:

```
for each database, db, specified on the command line {

  for each segment, seg, in db {

    // plan mode
    for each top-level object, o, in seg
      Invoke the planner for o's type on o

    // dump mode
    for each top-level object, o, in seg
      Invoke the object dumper for o's type on o
      If necessary, create a fixup dumper for o, and
      associate it with either seg, db, or the whole dump

    // fixup mode
    for each fixup dumper associated with seg
      Invoke that fixup dumper
  }

  // fixup mode
  for each fixup dumper associated with db
    Invoke that fixup dumper
}

// fixup mode
for each fixup associated with the whole dump
  invoke that fixup dumper
```

By default, object-dump actions sometimes invoke other object-dump actions on embedded objects. The following summarizes the behavior of the different object dump actions invoked by the default dumper for different types of objects:

- If the object is a fundamental value, pointer, or C++ reference, a type-specific dumper is invoked that dumps the value by using the C++ stream operator.
- If the object is an array, the array dumper, which handles these cases recursively for each array element, is invoked.
- If the object is an instance of a class, a class-specific dumper is invoked, if there is one. Otherwise, the generic class-instance dumper is invoked, which handles these cases recursively for each data member and base class of the class.
- If the object is an instance of an ObjectStore class, it invokes a class-specific dumper.

Default fixup-dump actions also invoke fixup-dump actions on embedded objects in the same way.

For each object form the loader processes, it invokes a type-specific object loader in a manner similar to that described for the dumper.

Supplying Customized Type-Specific Actions

To customize the dumping of objects of a given type, you specialize base classes whose instances represent the three kinds of dumper actions:

- Planner classes: one or more subclasses of `os_Planning_action` that handle identification of nonroot objects
- Object-dumper class: a subtype of `os_Dumper_specialization` that handles generation of object forms
- Fixup-dumper class: a subtype of `os_Fixup_dumper` that handles generation of fixup forms

To support planning for a given class, *`class`*, choose one of the following approaches:

- Shallow approach: for each type of nonroot object associated with instances of *`class`*, derive a class from `os_Planning_action`. For example, if you are customizing the dump of instances of `my_hash_table`, derive a class corresponding to each class of nonroot object that forms a hash table, such as `my_hash_table_slot` and `my_hash_table_overflow_list`.
- Deep approach: derive one class from `os_Planning_action`. An instance of this derived class serves as the planner for *`class`*. For example, if you are customizing the dump of instances of `my_hash_table`, derive the class `my_hash_table_planner`.

The invocation operator of a planner corresponding to a given class takes an instance of the class as argument. For example, `my_hash_table_planner::operator ()()` takes an instance of `my_hash_table` as an argument. With the deep approach, the invocation function typically navigates from the root object to the nonroots and creates an *ignore record* for each nonroot object.

With the shallow approach, the invocation function creates an ignore record for the argument or does nothing, depending on whether the argument is a nonroot object of the data structure whose dump is being customized.

The shallow approach has better paging behavior, so use it if possible.

For each derived class supporting object dumping, planning, and fixup dumping, perform these steps:

- Declare the derived class with certain members (see the following table).
- Implement the members.

In addition, for each derived class supporting object dumping and planning, perform these steps:

- Define an instance (planner and object dumper only).
- Register the instance (planner and object dumper only).

The following table shows the members of each class that you should implement. Your implementations of these functions specify the objects to be ignored and the dump formats.

Base Class	*Members*
`os_Planning_action`	`operator ()()`
`os_Dumper_specialization`	`operator ()()`
	`should_use_default_constructor()`
	`get_specialization_name()`
`os_Fixup_dumper`	Constructor
	`dump_info()`
	`duplicate()`

Build the dumper executable, `osdump`, using a copy of `/etc/dumpload/makefile.w32` (for Windows platforms) or `makefile.unx` (for UNIX platforms) in your installation directory. Be sure to set all the macros whose name begins with `USER_`.

Customizing Loads

During a load, the loader processes object and fixup forms in the order in which they were emitted by the dumper. To provide a customized loader for a given type, you implement functions that support the following tasks:

- Translation of an object form for the given type into the creation of a root object
- Translation of a fixup form for the given type into operations that modify the root object or recreate the nonroot portion of the object

To do this, derive a class from each of the following base classes:

- `os_Type_loader` (handles object form translation)
- `os_Type_info` (holds information about the load of the current object form)
- `os_Type_fixup_loader` (handles fixup form translation)
- `os_Type_fixup_info` (holds information about the load of the current fixup form)

For each derived class, you must perform these steps:

- Declare the class with certain members (see the following table).
- Implement the members.
- Define an instance.
- Register the instance.

The following table shows the members of each class that you should implement:

Base Class	*Members*
`os_Type_info`	`data`
	Constructor
`os_Type_loader`	`operator ()()`
	`load()`
	`create()`
	`fixup()`
	`get()`
`os_Type_fixup_info`	`fixup_data`
	Constructor
`os_Type_fixup_loader`	`operator ()()`
	`load()`
	`fixup()`
	`get()`

Build the loader executable, `osload`, using a copy of `/etc/dumpload/makefile.w32` (for Windows platforms) or `makefile.unx` (for UNIX platforms) in your installation directory. Be sure to set all the macros whose name begins with `USER_`.

Specializing os_Planning_action

Your specialization of `os_Planning_action` handles planning, including the identification of objects for which object forms should not be generated.

If you are using the shallow approach to planning, for each type, *`type`*, of nonroot object you must define a class that is

- Named *`type`*`_planner`
- Derived from `os_Planning_action`

For example, if `my_table_entry` is a type of nonroot object, use the following:

```
class my_table_entry_planner : public os_Planning_action {...}
```

If you are using the deep approach to planning, for the type, *`type`*, of the root object, you must define a class that is

- Named *`type`*`_planner`
- Derived from `os_Planning_action`

For example, if `my_hash_table` is a type whose dump and load you want to customize, use the following:

```
class my_hash_table_planner : public os_Planning_action {...}
```

The derived class must implement `operator ()()`.

You must also define and register an instance of the derived class.

If a class does not require fixups, it does not require planning. Even if it does require fixups, it might not require planning. If the fixups do not create any objects that would be dumped normally without planning, planning might be unnecessary. However, if you want a dumper to *not* dump certain objects, for example, the referent of a pointer for a given class, you should create a planner for the given class that will mark the referent to be ignored.

Implementing operator ()()

```
void type_planner::operator () (
  const os_type& actual_type,
  void* object
);
```

Using shallow approach

If you are taking the shallow approach to planning, implement *type*`_planner::operator ()()` (where *type* is one type of nonroot object) to do the following:

- Do type verification (optional). The *actual_type* argument is supplied for this purpose.
- Dereference *object* and cast the result to `type&`.
- Determine whether the object should be ignored during object-dump mode. This is entirely application specific.

If the object should be ignored, do the following:

- Create a stack-based `os_Dumper_reference` to the object. Use `os_Dumper_reference::os_Dumper_reference(void*)`. For more information, see `os_Dumper_reference::os_Dumper_reference()` in Chapter 2 of the *C++ API Reference*.
- Retrieve the database table. Use `os_Database_table::get()`.
- Insert the `os_Dumper_reference` into the database table. Use `os_database_table::insert(os_Dumper_reference&)`. For more information, see `os_Database_table::insert()` in Chapter 2 of the *C++ API Reference*.

Example

Following is a typical implementation:

```
void type_planner::operator () (
  const os_type& actual_type,
  void* object
)
{
  ...
  /* Do type verification (optional) */
  assert_is_type(actual_type, object);
  ...
  type& obj = (type&)*object;
  ...

  if (should_ignore(obj)) {
    Dumper_reference ignored_object(&obj);
```

```
    Database_table::get().insert(ignored_object);
  }
  ...
}
```

In this example, `assert_is_type()` and `should_ignore()` are user defined.

If necessary, you can include application-specific processing in place of the "..."s.

Using deep approach

If you are using the deep approach to planning, implement *type*`_:oplanner::operator ()()` (where *type* is the type of the root object) to do the following:

- Do type verification (optional). The `actual_type` argument is supplied for this purpose.
- Dereference `object` and cast the result to `type&`.
- Retrieve the database table. Use `os_Database_table::get()`. For more information about this function, see `os_Database_table::get()` in Chapter 2 of the *C++ API Reference*.
- Find the associated nonroot objects.

For each associated nonroot object, do the following:

- Create a stack-based `os_Dumper_reference` to the object or set one to refer to the object. Use `os_Dumper_reference::os_Dumper_reference(void*)` or `os_Dumper_reference::operator =()`.
- Insert the `os_Dumper_reference` into the database table. Use `os_Database_table::insert(os_Dumper_reference&)`. For more information about this function, see `os_Database_table::insert()` in Chapter 2 of the *C++ API Reference*.

Example

Following is a typical implementation:

```
{
  ...
  /* Do type verification (optional) */
  assert_is_type(actual_type, object);
  ...
  type& obj = (type&)*object;
  ...
  if (should_ignore(obj) {
    Dumper_reference ignored_object(&obj);
    Database_table::get().insert(ignored_object);
  }
  (*reachable_type_planner)(obj.reachable_pointer);
  ...
}
```

If you can define dump-related members of *type*, you can use the following approach:

```
void <type>_planner::operator () (const os_type& actual_type,
      void* object)
{
  ...
  type& obj = (type&)*object;
```

```
  obj.plan_dump();
  ...
}

void type::plan_dump ()
{
  ...
  if (should_ignore(obj) {
    Dumper_reference ignored_object(&obj);
    Database_table::get().insert(ignored_object);
  }
  ...
  /* consider directly reachable objects */
  reachable_pointer->plan_dump();
  ...
}
```

Defining and Registering the Instance

You must define an instance:

```
static type_planner the_type_planner;
```

and register it by including an entry in the global array `entries[]`:

```
static os_Planner_registration_entry planner_entries[] = {
  ...
  os_Planner_registration_entry("type", &the_type_planner),
  ...
}

static const unsigned number_planner_registration_entries
  = OS_NUMBER_REGISTRATIONS(
    planner_entries,
    os_Planner_registration_entry
);

static os_Planner_registration_block planner_block(
  planner_entries,
  number_planner_registration_entries,
  __FILE__,
  __LINE__
);
```

This code should be at top level.

Specializing os_Dumper_specialization

Your specialization of `os_Dumper_specialization` handles the dumping of object forms.

To customize the dump and load of instances of a type, *type*, define a class that is

- Named *type*`_dumper`
- Derived from `os_Dumper_specialization`

For example, to customize the dump and load of instances of `my_collection`, use the following:

```
class my_collection_dumper : public os_Dumper_specialization {...}
```

The derived class must implement the following functions:

- `operator ()()`: handles generation of the value portion of object forms
- `should_use_default_constructor()`: determines the constructor that is called in dumper-generated code for constructing embedded objects

Under some circumstances, you must implement *`type`*`_dumper::get_specialization_name()`.

You must also define and register an instance of the derived class.

Implementing operator ()()

```
void type_dumper::operator () (
  const os_class_type& actual_class,
  void* object
  );
```

An object form has the following structure:

```
id (Type) value
```

When you supply an object-form dumper, you are responsible for generation of the *`value`* portion only. The functions that generate the rest of the object form are inherited from `os_Dumper_specialization`.

The *`value`* portion is generated by `operator ()()`. Implement `operator ()()` to generate ASCII from which a loader can reconstruct function arguments. The arguments should be those required for recreation of the root portion of the object being dumped.

Define `operator ()()` to do the following:

- Dereference the `void*` argument and cast the result to *`type`*. (Optionally, do type verification first.)
- Output the *`value`* portion of the object form.
- Create a *`type`*`_fixup_dumper` on the stack, passing the `void*` argument and the `os_class_type&` argument to the *`type`*`_fixup_dumper` constructor.
- Insert the *`type`*`_fixup_dumper` into the database table.

Inserting a fixup dumper causes the dumper to generate a fixup form for *`object`* after generating all the object forms. When a load processes this kind of fixup form for *`object`*, it adjusts all pointers and C++ references in *`object`* so that they refer to the appropriate newly loaded referent.

If *`type`* is a nonarray type, the typical implementation has the following form:

```
void type_dumper::operator () (
  const os_class_type& actual_class,
  void* object
)
```

```
{
  ...
  // optional type verification
  // assert function defined by user
  assert_is_type(actual_class, object);
  ...
  // cast the void*
  type& obj = (<Type&>)*object;
  ...
  // output the value portion of the object form
  get_stream() << obj.get_representation() << ' '
    << obj.get_size() << ' ';
  ...
  // create a Fixup_dumper on the stack
  type_fixup_dumper fixup(
    get_stream(),
    *this,
    actual_class,
    object
);
  // insert a fixup dumper for processing at the end of the dump
  Database_type::get().insert(fixup);
}
```

This example assumes that the output of *type*`::get_representation()` and *type*`::get_size()` provides sufficient information to create the root of `object`.

You can insert application-specific processing in place of the "..."s, but this is not required.

You usually need not customize array types because you can customize the element type of an array type. The default array dumper calls the custom dumper on each array element.

If you do want to customize the array dumper, implement the following overloading of the invocation operator:

```
void type_dumper::operator () (
  const os_class_type& actual_class,
  void* object,
  unsigned number_elements
)
{
  ...
}
```

Implementing should_use_default_constructor()

```
os_boolean should_use_default_constructor(
  const os_class_type& class_type
) const;
```

If there are *no* instances of *type* embedded (as a data member value or object corresponding to a base class) in instances of a noncustomized class, define this function to return `1`. If there are instances of *type* embedded in instances of a noncustomized class, you might want to define this function to return `0`.

When you customize the dump and load of a type, you supply code to create instances of the type during a load. See Implementing create() on page 190. This code

is used for all nonembedded instances of the type. For an instance of the type embedded in a noncustomized type, the loader calls the customized type's constructor automatically, using code generated by the dumper.

For a given customized type, *type*, you determine which of two constructors is called in the dumper-generated code:

- The no-argument constructor, *type*::*type*()
- The special loader constructor *type*(*type*_data&)

See Implementing data on page 187 for information on *type*_data.

If you implement type_Loader::should_use_default_constructor() to return 0, the dumper-generated code calls the no-argument constructor. If you implement type_Loader::should_use_default_constructor() to return 1, the dumper-generated code calls the special loader constructor.

If there are nonembedded instances of *type*, the constructor you specify with should_use_default_constructor() must be the same as the constructor you call in the corresponding loader's create() function. See Implementing create() on page 190 for more information.

Implementing get_specialization_name()

```
char* get_specialization_name(
  const os_class_type& class_type
) const;
```

You must define this function only if there is a subtype of *type* such that both of the following hold:

- In the database to be dumped, an instance of the subtype is embedded in another object.
- The subtype is not customized.

In this case, define the function to return the character string "*type*", given an os_class_type& for each such subtype.

Deleting the returned string is the responsibility of the caller of this function.

Defining and Registering the Dumper Instance

You must define an instance of type_dumper:

```
static type_dumper the_type_dumper;
```

and register it by including an entry in the global array entries[]:

```
static os_Dumper_registration_entry entries[] = {
  ...
  os_Dumper_registration_entry("type", &the_type_dumper),
  ...
};

static const unsigned number_dumper_registration_entries
  = OS_NUMBER_REGISTRATIONS(
    entries,
```

```
    os_Dumper_registration_entry
  );

static os_Dumper_registration_block block(
  entries,
  number_dumper_registration_entries,
  __FILE__,
  __LINE__
);
```

This code should be at top level.

Specializing os_Fixup_dumper

The dumper invokes type-specific fixup-dump actions that typically emit strings from which the loader can reconstruct arguments. Your specialization of `os_Fixup_dumper` handles the dumping of fixup forms.

To customize the dump and load of instances of a type, *`type`*, define a class that is

- Named *`type`*`_fixup_dumper`
- Derived from `os_Fixup_dumper`

For example, to customize the dumping and loading of instances of `my_collection`, use the following:

```
class my_collection_fixup_dumper :
  public os_Fixup_Dumper {...}
```

The derived class must implement the following functions:

- `dump_info()`: handles generation of the value portion of fixup forms
- `duplicate()`: supports insertion of an instance of *`type`*`_fixup_dumper` into the database table
- Constructor: passes arguments to the base type constructor

For a description of the `os_Fixup_dumper` class and its member functions, see `os_Fixup_dumper` in Chapter 2 of the *C++ API Reference*.

Implementing dump_info()

```
void type_fixup_dumper::dump_info()
```

A fixup form has the following structure:

```
fixup id (Type) info
```

When you supply a fixup-form dumper, you are responsible for generation of the *`info`* portion only. The functions that generate the rest of the fixup form are inherited from `os_Fixup_dumper`.

The *`info`* portion is generated by `dump_info()`. Implement `dump_info()` to generate ASCII from which a loader can reconstruct function arguments. The arguments

should be those required for recreation of the nonroot portion of the object being fixed up.

Where *type* is the type of object being fixed up, the function should

- Do type verification (optional). Retrieve the type of the object being fixed with `os_Fixup_dumper::get_type()`. For information about this function, see `os_Fixup_dumper::get_type()` in Chapter 2 of the *C++ API Reference*.
- Retrieve the object being fixed with `os_Fixup_dumper::get_object_to_fix()`. Cast the result to *type*`&`. For information about this function, see `os_Fixup_dumper::get_object_to_fix()` in Chapter 2 of the *C++ API Reference*.
- Output the strings from which the loader can reconstruct the arguments.

To dump arguments that are pointers or C++ references, use the `os_Dumper_reference` class as follows:

- Use `os_Dumper_reference::operator =()` to create a stack-based dumper reference corresponding to the pointer or C++ reference. For information about this function, see `os_Dumper_reference::operator =()` in Chapter 2 of the *C++ API Reference*.
- Pass the dumper reference to `operator <<()`, to add its ASCII to the dump stream.

For information about this function, see `os_Dumper_reference` in Chapter 2 of the *C++ API Reference*.

A loader can reconstruct the dumper reference from the load stream with `operator >>()` and get the location of the newly loaded referent with `os_Dumper_reference::resolve()`. For more information about this function, see `os_Dumper_reference::resolve()` in Chapter 2 of the *C++ API Reference*.

Example

For example, a fixup form for a collection might include ASCII from which a dumper can reconstruct all the collection's elements:

```
void type_fixup_dumper::dump_info() const
{
  ...
  const void* object = get_object_to_fix();
  assert_is_type(get_type(), object);
  ...
  type& obj = (type&)*object;
   ...
  os_Dumper_reference ref;
  for (unsigned count = 0; count < obj.get_size(); ++count) {
    element_type& element = obj[count];
    ref = element;
    get_stream() << ref << ' ';
  }
  // info terminator -- assumes no null elements
  ref = 0;
  get_stream() << ref << ' ';
  ...
}
```

In this example,

- *element_type* is the type of object contained by instances of *type* (the example assumes instances of *type* are collections).
- assert_is_*type*() *type*::operator []() and *type*::operator []() are user defined.

If necessary, you can include additional application-specific processing in place of the "..."s.

Implementing duplicate()

```
Fixup& type_fixup_dumper::duplicate()
```

Implement duplicate() to allocate a copy of this *type*_fixup_dumper in the specified segment. The following example assumes that *type*_fixup_dumper defines a copy constructor and a get_os_typespec() function.

```
Fixup& type_fixup_dumper::duplicate(
  os_segment& segment
) const
{
  return *new(segment, type_fixup_dumper::get_os_typespec())
    type_fixup_dumper(*this);
}
```

Be sure to include *type*_fixup_dumper in the application schema of the emitted loader.

Implementing the Constructor

```
type_fixup_dumper(
  os_Dumper_stream&,
  os_Dumper&,
  const os_class_type&,
  const os_Dumper_reference object_to_fix,
  unsigned number_elements = 0
);
```

Implement the constructor to pass the arguments to the base type constructor:

```
os_Fixup_dumper(
  os_Dumper_stream&,
  os_Dumper&,
  const os_class_type&,
  const os_Dumper_reference object_to_fix,
  unsigned number_elements = 0
);
```

Specializing os_Type_info

```
class type_info : public os_Type_info
```

A *type*_info holds information about the loading of the object form currently being processed, as described in os_Type_info in *C++ API Reference*, Chapter 2. The derived type must:

- Declare the data member *type*_info::data: this member points to an instance of *type*_data, which holds the information required to construct the object being loaded.
- Define a *type*_info constructor: the constructor takes a *type*_data argument; passes other arguments to a base type constructor.
- Declare *type*_loader as a friend class of *type*_info.

For each instance of *type* being loaded, *type*_loader::operator ()() makes an instance of *type*_data and an instance of *type*_info that points to the *type*_data. It passes the *type*_data to load(), which sets the fields of the *type*_data based on the contents of the dump stream.

*type*_loader::operator ()() then passes the *type*_info to create(), which uses the information to create the postload object.

Your specialization can add any members you find convenient.

Implementing data

*type*_data &data;

To implement this public data member, derive a type, *type*_data, from os_Type_data (this base class has no members). Define a data member of *type*_data for each portion of the object-form value emitted by *type*_dumper::operator ()(). These members are set by load() based on information in the load stream and are used later by create() and fixup() to recreate the object being loaded.

*type*_data should have a data member for each data member and base class of *type*. For pointer and (C++) reference members of *type*, the *type*_data member should be of type os_Fixup_reference. For embedded arrays, the *type*_data member should also be of type os_Fixup_reference. For embedded class members, the type should be *embedded_class*_data. All data members corresponding to a base class should have the base class as their value type.

All other data members should have the same value types as the corresponding members of *type*. All data members of *type*_data should have the same names as the corresponding members or base classes of *type*.

Implementing the Constructor

```
type_info(
  type_loader& cur_loader,
  os_Loader_stream& lstr,
  os_Object_info& info,
  type_data& data_arg
);
```

Implement this function to set *type*_info::data to *data_arg*. Pass the first three arguments to the following os_Type_info constructor:

```
os_Type_info(
  os_Type_loader& cur_loader,
  os_Loader_stream& lstr,
  os_Object_info& info
);
```

cur_loader is the loader for the object currently being loaded.

lstr is the dump stream from which the current object form is being processed.

info is the loader information for the object being loaded.

When you call *type*_info::*type*_info() from within *type*_loader::operator ()(),

- Pass *this as *cur_loader*.
- Pass the stream passed in (as *lstr*) to operator ()().
- Cast to os_Object_info& the loader information passed in to operator ()(). Pass the result of the cast as *info*.

You can define this constructor to have additional arguments if they are necessary for your specialization.

For more information about the constructor, see os_Type_info::os_Type_info() in Chapter 2 of the *C++ API Reference* .

Specializing os_Type_loader

Your specialization of os_Type_loader handles the loading of object forms.

To customize the dumping and loading of instances of a type, *type*, define a class that is

- Named *type*_loader
- Derived from os_Type_loader

For example, to customize the dumping and loading of instances of my_collection, use the following:

```
class my_collection_loader :
  public os_Type_loader {...}
```

The derived class must implement the following functions:

- operator ()(): calls load() and create().
- load(): reads the value portion of an object form from the load stream.
- create(): creates the postload object based on information set by load(). Also calls fixup().
- fixup(): inserts fixup records into the database table for pointer or reference adjustment.
- get(): returns the one and only instance of the derived class.

You must also define and register an instance of the derived class.

Implementing operator ()()

```
os_Loader_action* type_loader::operator () (
  os_Loader_stream& stream,
  os_Loader_info& previous_info
)
```

Implement the invocation operator to do the following:

- Create an instance of *type*_data on the stack.
- Create a stack-based instance of *type*_info, passing the *type*_data and *previous_info* to the *type*_info's constructor. Cast *previous_info* to os_Object_info& before passing it to the os_Type_info() constructor.
- Pass the *type*_data to load().
- Pass the *type*_info to create().

Example

Following is an example:

```
Loader_action* type_loader::operator () (
  os_Loader_stream& stream,
  os_Loader_info& previous_info
)
{
  os_Object_info& object_info = previous_info;
  type_data data;
  type_info info(*this, stream, object_info, data);
  load(stream, data, 1);
  create(info);
  return 0;
}
```

If there are instances of *type* embedded in instances of a noncustomized type, this function is not called on those embedded instances. Instead, the loader calls your load() and fixup() functions directly. Your create() function is not called for embedded instances; the loader uses a generated constructor call instead.

Implementing load()

```
Loader_action* type_loader::load(
  os_Loader_stream& stream,
  os_Type_data& given_data,
  os_boolean is_most_derived_class
)
```

This function is responsible for reading the value portion of the object form from the load stream and setting the data members of *given_data* accordingly.

After load() sets the data members of *given_data*, create() or fixup() uses *given_data* to guide recreation of the object being loaded.

The function should do the following:

- Cast *given_data* to *type*_data&.
- Input each part of the dumped value; use the current portion of the dumped value to set a data member of *type*_data. This value is used by *type*_loader::create() to create the object being loaded.

Returns `0` for success.

Example

Following is an example:

```
Loader_action* type_loader::load(
  os_Loader_stream& stream,
  os_Type_data& given_data,
  os_boolean is_most_derived_class
)
{
  type_data& data = (type_data&) given_data;
  ...
  /* Input each part of the dumped value. */
  stream >> data.representation;
  stream >> data.size;
  ...
  return 0;
}
```

If the current portion of the dumped value is an embedded object form for a class, *embedded_class*, retrieve that class's loader with *embedded_class*`_loader::get()` and call *embedded_class*`_loader::load()`, passing `stream` and the *embedded_class*`_data` embedded in *type*`_data`:

```
/* Load embedded class. */
embedded_type_loader::get().load(
  stream,
  data.member,
  1
);
```

Implementing create()

```
void type_loader::create(os_Loader_info& given_info) ;
```

This function is responsible for creating the persistent object corresponding to the object form being loaded. Arguments to persistent `new` can be retrieved from *given_info*. Arguments to the object constructor can be retrieved from the *type*`_data` associated with *given_info*.

The function should

- Cast *given_info* to *type*`_info&`.
- Call `os_Type_info::get_replacing_location()` on the result of the cast. For more information about this function, see `os_Type_info::get_replacing_location()` in Chapter 2 of the *C++ API Reference*.
- If the result is nonzero, the object being created is an element of a top-level array. Call top-level operator `new` with the result of `get_replacing_location()` as placement argument.
- If the result of `get_replacing_location()` is `0`, the object is not being created as an element of a top-level array. Call persistent `new` with the result of `os_Type_info::get_replacing_segment()` as the placement argument; see `os_Type_info::get_replacing_segment()` in Chapter 2 of the *C++ API Reference*.

- In either case, call a type constructor that restores all or part of the state of the object being loaded. This constructor must not affect any state outside of the object being loaded.

If the constructor call did not restore the entire state of the object, you must define `fixup()` to restore the remaining state.

If there are instances of *`type`* embedded (as a data member value or object corresponding to a base type) in an instance of a noncustomized type, the constructor called in your `create()` function must be either a no-argument constructor or the special loader constructor, *`type`*`(`*`type`*`_data&)`.

If you use the special loader constructor, define *`type`*`_dumper::should_use_default_constructor()` to return `0`.

If you use a no-argument constructor, define *`type`*`_dumper::should_use_default_constructor()` to return `1`.

The function must also create a mapping record that records the predump and postload locations of the object being loaded. To do this, the function must

- Get the type of object being loaded by performing `os_Type_info::get_type()` on *`given_info`*. For more information about this function, see `os_Type_info::get_type()` in Chapter 2 of the *C++ API Reference*.
- Construct a stack-based `os_Dumper_reference` corresponding to the postload location of the object being loaded. Pass the location of the newly loaded object (that is, the pointer returned by `new`) to the constructor.
- Call `os_Type_info::set_replacing_location()` on the *`type`*`_info`, passing the location of the newly loaded object as argument. For more information about this function, see `os_Type_info::set_replacing_location()` in Chapter 2 of the *C++ API Reference*.
- Get the original (that is, predump) location of the loaded object with `os_Type_info::get_original_location()`. For more information about this function, see `os_Type_info::get_original_location()` in Chapter 2 of the *C++ API Reference*.
- Retrieve the database table by using `os_Database_table::get()`. For more information about this function, see `os_Database_table::get()` in Chapter 2 of the *C++ API Reference*.
- Call `os_Database_table::insert()` on the database table, passing the original location, the dumper reference, and the type of the dumped object. For more information about this function, see `os_Database_table::insert()` in Chapter 2 of the *C++ API Reference*.

Finally, the function must call *`type`*`_loader::fixup()`, passing as arguments the *`type`*`_data` and the location of the newly loaded object (that is, the pointer returned by `new`).

Example

Following is an example:

```
void type_loader::create (os_Loader_info& given_info)
{
  type_info& info = (type_info&) given_info;
  type* value;
  void* location = info.get_replacing_location();

  if (location)
    value = ::new(location) type(info.data);

  else {
    value = new (
        &info.get_replacing_segment(),
        type::get_os_typespec()
      ) type(info.data);

// Insert a mapping of the constructed object's original
// location to its replacing location into the Database_table.
    const os_type& value_type = info.get_type();
    os_Dumper_reference replacing_location(value);
    info.set_replacing_location(value);
    os_Database_table::get().insert(
      info.get_original_location(),
      replacing_location,
      value_type
    );
  }
  fixup(info.data, value, 1);
}
```

Implementing fixup()

```
void type_loader::fixup(
  os_Type_data& given_data,
  void* object,
  os_boolean is_most_derived_class
)
```

The default loader automatically adjusts pointers and references in loaded objects. If you supply a type-specific loader, *type*_loader, you must explicitly direct the loader to make these adjustments for instances of *type*.

You do this by defining `fixup()` to insert fixup records into the database table. Perform one insert for each pointer, C++ reference, or ObjectStore reference contained directly within instances of *type*. For each one, do the following:

- Construct a stack-based `os_Dumper_reference` corresponding to the address of the pointer or reference contained in *object*.
- Construct a stack-based `os_Dumper_reference` corresponding to the predump value of the pointer or reference contained in *object*.
- Retrieve the database table by using `os_Database_table::get()`.
- Call `os_Database_table::insert()` on the database table, passing the enumerator `os_reference_fixup_kind::pointer` and the two dumper references.

`fixup()` is also responsible for setting members of the newly loaded object if the constructor called by `create()` does not restore the entire state of the object being loaded. In this case, make *`type`*`_loader` a friend of *`type`* to allow `fixup()` to access private members of *`type`*.

If the constructor called by `create()` *does* restore the entire state of the object being loaded, and instances of *`type`* contain no pointers, no C++ references, and no ObjectStore references, you need not implement this function. You never have to implement a no-op `fixup()`.

Example

Following is a typical implementation:

```
void type_loader::fixup (
  os_Type_data& given_data,
  void* object,
  os_boolean is_most_derived_class
)
{
  type_data& data = (type_data&) given_data;
  type& obj = type& *object;
  ...
  /* Fixup pointer. */
os_Dumper_reference pointer_location(&obj.pointer);
os_Dumper_reference original_referent_location(data.pointer);
  os_Database_table::get().insert
    (os_Reference_fixup_kind::pointer, pointer_location,
    original_referent_location);
  ...
}
```

If a portion of the dumped value is an embedded object form for a class, *`embedded_class`*, retrieve that class's loader with *`embedded_class`*`_loader::get()` and call *`embedded_class`*`_loader::fixup()`, passing the *`embedded_class`*`_data` embedded in *`type`*`_data` and the corresponding object embedded in `obj`:

```
  /* Fixup embedded object. */
  embedded_type_loader::get().fixup(
    data.member,
    &obj.member
  );
```

Implementing get()

Define a global variable whose value is an instance of *`type`*`_loader`. Define *`type`*`_loader::get()` to return this instance:

```
static type_loader the_type_loader;
```

```
Type_loader& type_loader::get()
{
  return the_type_loader;
}
```

Defining and Registering the Instance

You must define an instance:

```
static type_loder the_type_loader;
```

and register it by including an entry in the global array `entries[]`:

```
static os_Loader_registration_entry entries[] = {
  ...
  os_Loader_registration_entry("type", &the_type_loader),
  ...
}

static const unsigned number_loader_registration_entries =
  OS_NUMBER_REGISTRATIONS(
    entries,
    os_Loader_registration_entry
);

static os_Loader_registration_block block(
  entries,
  number_loader_registration_entries,
  __FILE__,
  __LINE__
);
```

This code should be at top level.

Specializing os_Type_fixup_info

```
class type_fixup_info : public os_Type_fixup_info
```

A *type*`_fixup_info` holds information about the loading of the fixup form currently being processed. `os_Type_fixup_info` is derived from `os_Type_info`; see `os_Type_info` in *C++ API Reference,* Chapter 2.

type`_fixup_info` must define two members:

- *type*`_fixup_info::fixup_data`: points to an instance of *type*`_fixup_data`, which holds the information required to construct the object being loaded.
- *type*`_fixup_info` constructor: passes arguments to a base type constructor.

For each instance of *type* being loaded, *type*`_fixup_loader::operator ()()` makes an instance of *type*`_fixup_data` and an instance of *type*`_fixup_info` that points to the *type*`_fixup_data`. It passes the *type*`_fixup_data` to `load()`, which sets the fields of the *type*`_fixup_data` based on the contents of the dump stream.

type`_loader::operator ()()` then passes the *type*`_fixup_info` to `fixup()`, which uses the information to create the postload object.

Your specialization can add any members you find convenient.

Implementing fixup_data

```
type_fixup_data &fixup_data;
```

To implement this public data member, derive a type, *type*_fixup_data, from os_Type_data. Define a data member of *type*_fixup_data for each portion of the object-form value emitted by *type*_fixup_dumper::dump_info(). These members are set by load() based on information in the load stream and are used later by fixup() to recreate the object being loaded.

Implementing the Constructor

```
type_fixup_info(
  os_Type_fixup_loader cur_fixup_loader,
  os_Loader_stream lstr,
  os_Object_info& info,
  type_fixup_data &data_arg
);
```

Implement this function to set *type*_fixup_info::data to data_arg. Pass the first three arguments to the following os_Type_fixup_info constructor:

```
os_Type_fixup_info(
  os_Type_loader& cur_loader,
  os_Loader_stream& lstr,
  os_Fixup_info& info
);
```

cur_loader is the loader for the object currently being fixed up.

lstr is the dump stream from which the current fixup form is being processed.

info is the fixup loader information for the object being fixed up.

When you call *type*_fixup_info::*type*_fixup_info() from within *type*_loader::operator ()()

- Pass *this as *cur_loader*.
- Pass the stream passed in (as *lstr*) to operator ()().
- Cast to os_Fixup_info& the loader information passed in to operator ()(). Pass the result of the cast as *info*.

You can define this constructor to have additional arguments if they are necessary for your specialization.

Specializing os_Type_fixup_loader

Your specialization of os_Type_fixup_loader handles the loading of fixup forms.

To customize the dump and load of instances of type *type*, define a class that is

- Named *type*_fixup_loader
- Derived from os_Type_fixup_loader

For example, to customize the dump and load of instances of `my_collection`, use the following:

```
class my_collection_fixup_loader :
  public os_Type_fixup_loader {...}
```

The derived class must implement the following functions:

- `operator ()()`: calls `load()` and `fixup()`.
- `load()`: reads the info portion of a fixup form from the load stream.
- `fixup()`: performs fixup based on information set by `load()`.
- `get()`: returns the one and only instance of the derived class.

You must also define and register an instance of the derived class.

Implementing operator ()()

```
os_Loader_action* type_fixup_loader::operator () (
  os_Loader_stream& stream,
  os_Loader_info& previous_info
)
```

Implement the invocation operator to do the following:

- Create an instance of *type*`_fixup_data` on the stack.
- Create a stack-based instance of *type*`_fixup_info`, passing the *type*`_fixup_data` and `previous_info` to the *type*`_fixup_info`'s constructor. Cast `previous_info` to `os_Fixup_info&` before passing it to the constructor.
- Pass the *type*`_data` to `load()`.
- Pass the *type*`_info` to `fixup()`.

This assumes that one call to `load()` consumes the entire info portion of a fixup form. You can also implement `load()` to consume only a part of the info portion and call `load()` and `fixup()` multiple times from `operator ()()`. The latter approach makes loaders more scalable for large info portions.

Example

Following are examples:

```
os_Loader_action* type_fixup_loader::operator () (
  os_Loader_stream& stream,
  os_Loader_info& previous_info
)
{
  os_Fixup_info& fixup_info = previous_info;
  type_fixup_data data;
  type_fixup_info info(*this, stream, fixup_info, data);
  load(stream, data, 1);
  fixup(info, 1);
  return 0;
}
```

You can also call `load()` and `fixup()` more than once:

```
os_Loader_action* type_fixup_loader::operator () (
  os_Loader_stream& stream,
  os_Loader_info& previous_info
```

```
)
{
  os_Fixup_info& fixup_info = previous_info;
  type_fixup_data data;
  for (;;) {
    type_fixup_info info(*this, stream, fixup_info, data);
    if( ! load(stream, data, 1))
      break;
    fixup(info, 1);
  }
  return 0;
}
```

This assumes `load()` is implemented to return `0` when the entire info portion has been consumed.

Implementing load()

```
Loader_action* load(
  Loader_stream& stream,
  Type_data& given_data,
  os_boolean is_most_derived_class
);
```

This function is responsible for reading the info portion of the fixup form from the load stream and setting the data members of *`given_data`* accordingly. If the invocation operator calls `load()` just once, `load()` must consume all the info portion. If the invocation operator calls `load()` multiple times, each call to `load()` must consume only part of the info portion.

After `load()` sets the data members of `given_data`, `fixup()` uses `given_data` to guide recreation of the nonroot portion of the object being fixed up.

The function should do the following:

- Cast *`given_data`* to *`type`*`_fixup_data&`
- Input each part of the dumped value; use the current portion of the dumped value to set a data member of *`type`*`_data`

If `load()` consumes the entire info portion, returns `0` for success.

If `load()` consumes only part of the info portion, you can return `0` when the entire info portion has been consumed and return nonzero otherwise. You must cast the return value to `os_Loader_action*`.

Example

Following is an example:

```
Loader_action* type_fixup_loader::load(
  Loader_stream& stream,
  Type_data& given_data,
  os_boolean is_most_derived_class
)
{
  type_fixup_data& data = (type_fixup_data&) given_data;
  ...
  /* Input one part of the dumped value. */
  os_Dumper_reference original_ref;
  stream >> original_ref;
```

```
  if (original_ref == 0) // info terminator
    return ( (os_Loader_action*) 0 );
  else {
    data.dumper_ref =
      os_Database_table::get().find_reference(original_ref);
    return ( (os_Loader_action*) 1 );
  }
}
```

If a portion of the dumped info is an embedded fixup form for a class, *embedded_class*, load it as follows:

- Retrieve that class's fixup loader with *embedded_class*_fixup_loader::get().
- Call *embedded_class*_fixup_loader::load(), passing stream and the *embedded_class*_fixup_data embedded in *type*_fixup_data.

For example:

```
/* Load embedded class. */
embedded_type_fixup_loader::get().load(
  stream,
  data.member_3,
  1
);
```

Implementing fixup()

```
void fixup(
  os_Type_fixup_info& info,
  os_boolean is_most_derived_class
);
```

This function performs fixup based on the information passed in. For example, if the object being fixed up is a collection and each portion of the fixup form identifies a collection element, fixup() inserts an element in the collection.

The function should

- Cast *info* to *type*_fixup_info&.
- Construct an os_Dumper_reference to the predump object.
- Construct a dumper reference to the postload object by performing os_Database_table::find_reference() on the dumper reference to the predump object. For more information about this function, see os_Database_table::find_reference() in Chapter 2 of the *C++ API Reference*.
- Cast the dumper reference to the postload object to *type**.
- Perform fixup on the postload object based on the information in *info*.

Example

Following is an example:

```
void type_fixup_loader::fixup (
  os_Type_fixup_info& given_info,
  os_boolean is_most_derived_class
)
{
  type_fixup_info& info = (type_fixup_info&) info;
```

```
  os_Dumper_reference original_location =
    info.get_original_location();
  if ( ! original_location) {
    // Handle error, there should only be a fixup for an
    // existing object.
  }

  os_Dumper_reference replacing_location =
   os_Database_table::get().find_reference(original_location);
  if ( ! replacing_location) {
    // Handle error, there should only be a fixup for an
    // existing object.
  }

  type* object = replacing_location;
  ...
/* Do whatever needs to be done to fix the designated object. */
  element_type *the_element = (element_type*) (
      info.fixup_data.dumper_ref.resolve()
    );
  object.insert(the_element)
  ...
}
```

Note that `info.fixup_data_.dumper_ref` is set by *type*`_fixup_loader::load()`.

Implementing get()

```
static os_Type_fixup_loader& get();
```

Define a global variable whose value is an instance of *type*`_loader`. Define the static function *type*`_loader::get()` to return this instance:

```
static type_fixup_loader the_type_fixup_loader;
os_Type_fixup_loader& type_fixup_loader::get()
{
  return the_type_fixup_loader;
}
```

Registering the Fixup Loader

Define an instance of the derived type:

```
static type_fixup_loader the_type_fixup_loader;
```

Register the instance of the derived class by including an entry in the global array `fixup_entries[]`:

```
static os_Fixup_registration_entry fixup_entries[] = {
  ...
  os_Fixup_registration_entry("type", &the_type_fixup_loader),
  ...
};

static const unsigned number_fixup_registration_entries =
  OS_NUMBER_REGISTRATIONS(
    fixup_entries,
    os_Fixup_registration_entry
);

static os_Fixup_registration_block fixup_block(
  fixup_entries,
  number_fixup_registration_entries,
  __FILE__,
  __LINE__
);
```

Chapter 7
Advanced Schema Evolution

This chapter provides information about the ObjectStore schema evolution facility. For a basic understanding of tasks that you must perform to complete a schema evolution project, see Chapter 9, Schemas, in the *C++ API User Guide*.

The information about schema evolution is organized in the following manner:

Phases of the Schema Evolution Process

The schema evolution process has two phases:

- *Schema modification*: modification of the schema information associated with the databases being evolved
- *Instance migration*: modification of any existing instances of the modified classes

(In this chapter, the term *process* is used in the ordinary nontechnical sense. The phrase *schema evolution process* refers to what the evolution facility does when it is invoked. This is not a system process separate from the execution of the application that calls the evolution function.)

Instance migration itself has two phases:

- Instance initialization
- Instance transformation

Instance Initialization

Instance initialization modifies existing instances of modified classes so that their representations conform to the new class definitions. This might involve adding or deleting fields or subobjects, changing the type of a field, or deleting entire objects. This phase of migration also initializes any storage components that have been added or that have changed type.

In most cases, new fields are initialized with zeros. There is one useful exception to this, however. In the case in which a field has changed type and the old and new types are assignment compatible, the new field is initialized by assignment from the old field value.

The initialization rules are discussed in Instance Initialization Rules on page 226.

Pointers to Modified Objects and Their Subobjects

During schema evolution, the addresses of objects generally change as they are moved to new locations. This is true for all objects, whether or not they are being migrated.The original objects and the evolved version of them will be in the same segment, but their cluster and/or their offset within their cluster might be different. If an object's size changes from less than 64 KB to 64 KB or greater, it can move to another cluster in the same segment.

Because of this, the schema evolution facility automatically translates all pointers to the instance so that they point to the new modified instance. This is done for all pointers in the databases being evolved, including pointers contained in instances of unmodified classes, cross-database pointers, and pointers to subobjects of migrated instances.

Untranslatable Pointers

During this process of adjusting pointers to modified instances, ObjectStore might detect various kinds of untranslatable pointers. For example, it might detect a pointer to the value of a data member that has been removed in the new schema. Because the data member has been removed, the subobject serving as the value of that data member is deleted as part of instance initialization. Any pointer to such a deleted subobject is untranslatable and is detected by ObjectStore.

In such a case, you can provide a special handler object to process the untranslatable pointer. Handler objects are derived from the class `os_untranslatable_pointer_handler`. Each time an untranslatable pointer is detected, the handler function is executed on the pointer (for example, the handler can change the untranslatable pointer to null or simply report its presence) and then schema evolution is resumed. If you do not provide a handler object, an exception is signaled by default when an untranslatable pointer is encountered.

C++ References

C++ references are treated as a kind of pointer. References to migrated instances are adjusted exactly as described previously. Untranslatable references are detected and can be handled as described.

ObjectStore References

In addition, as with pointers, ObjectStore references to migrated instances are adjusted to refer to the new instance rather than the old.

Untranslatable references are handled exactly the same as untranslatable pointers. ObjectStore references are just soft pointers.

Untranslatable pointers and references as well as untranslatable pointer and reference handlers are described in detail in Untranslatable Pointers on page 220.

Obsolete Indexes and Queries

Just as some pointers and references become obsolete after schema evolution, so do some indexes and persistently stored queries. For example, the selection criterion of a query or the path of an index might refer to a removed data member. ObjectStore detects all such queries and indexes. In the case of an obsolete query, ObjectStore by default signals an error that aborts schema evolution. However, you can provide an obsolete query handler function that internally marks the query so that schema evolution can continue.

You can handle obsolete queries or indexes by providing a special handler function for each. An obsolete index handler, for example, might create a new index using a path that is valid under the new schema. If you do not supply handlers, ObjectStore signals an exception when an obsolete query or index is encountered.

Handlers for obsolete indexes and queries are discussed in Obsolete Index and Query Handlers on page 224.

Task List Reporting

To help you get an overall picture of the operations involved in instance initialization for a particular evolution, the schema evolution facility allows you to obtain a task list describing the process. The task list consists of function definitions indicating the way the migrated instances of each modified class are to be initialized. You generate this list without actually invoking evolution, which allows you to verify your expectations concerning a particular schema change before migrating the data.

Task list reporting is discussed in Task List Reporting on page 225.

Instance Transformation

For some schema changes, the instance initialization phase is all that is needed. In other cases, however, further modification of class instances or associated data structures is required to complete the schema evolution. This further modification is generally application dependent, so ObjectStore allows you to define your own functions, *transformer functions*, to perform the task.

Transformer Functions

You associate exactly one transformer with each class whose instances you want to be transformed. During the transformation phase of instance migration, the schema evolution facility invokes each transformer function on each instance of the function's associated class, including instances that are subobjects of other objects. Transformer functions are associated with specific classes using the function `os_schema_evolution::augment_pre_evol_transformers()` or `os_schema_evolution::augment_post_evol_transformers()`. Use `augment_pre_evol_transformers()` to modify objects before any objects are moved or evolved; use `augment_post_evol_transformers()` to modify objects after all objects have been moved and evolved.

Transformers are useful for updating data structures that depend on the addresses of migrated instances. A hash table, for example, that hashes on addresses should be rebuilt using a transformer. Note that you need not rebuild a data structure if the position of an entry in the structure does not depend on the address of an object pointed to by the entry but depends instead, for example, on the value of some field of the object pointed to. Such data structures are still correct after the instance initialization phase.

Using transformers is discussed in Using Transformer Functions on page 209.

Initiating Schema Evolution

To perform schema evolution, you make and execute an application dedicated to the schema evolution process; the schema evolution API should not be used in an application that is used for normal database access. To perform schema evolution,

the schema application must invoke the member functions of os_schema_evolution in the following order:

1. Call os_schema_evolution::initialize() first before using any transient collections or ObjectStore features.
2. Call the various os_schema_evolution::set_*xxx* functions that are appropriate for your schema evolution. Examples are set_address_space_release_interval() and set_untranslatable_pointer_handler(). You also specify transformer functions during this step.
3. Specify the name of the application schema, if necessary. This is the new schema and is specifyied by making a call to os_schema_evolution::set_evolved_schema_db_name() and/or os_schema_evolution::augment_dll_schema_db_names(). If you do not call either of these functions, the application schema used is the application schema of the schema evolution application itself.
4. Call os_schema_evolution::evolve() to initiate the schema evolution.

   The evolve() function has two overloadings, declared as follows:

```
static void evolve(
  const char *workdb_name,
  const char *db_to_evolve
);

static void evolve(
  const char *workdb_name,
  const os_collection &dbs_to_evolve
);
```

   The evolution process depends on three parameters:

   - Databases to evolve
   - Schema modifications
   - Work database

5. Call os_schema_evolution::shutdown()

These functions must be called outside the dynamic scope of a transaction. The application must include the header file ostore/schmevol.hh and link with the libraries listed in the following table. Note that for Windows there is only one library, ostore.lib.

Library	*UNIX Link Option*	*Windows Library*
Metaobject Protocol	-losmop	ostore.lib
Schema Evolution	-losse	ostore.lib
Collections	-loscol	ostore.lib
Queries	-losqry	ostore.lib
Basic ObjectStore Libraries	-los	ostore.lib
Threads	-losthr	ostore.lib

Databases to Evolve

You specify the database or databases to be evolved as the second argument to `evolve()`. If you are evolving a single database, you supply a `char*` that denotes the pathname of the database. If you are evolving more than one database, you supply an `os_collection&` or `os_Collection<char*>&` that denotes a set containing the pathnames of the databases.

If you do not specify any database to evolve (that is, if you supply `0` for the first overloading, or an empty collection for the second overloading), `err_schema_evolution` is signaled.

The schema modifications are, by default, specified by the schema of the application that calls `evolve()`. Thus, the schema source file for this executable should contain a new class definition for each class that you want to modify.

If you want, you can specify the schema modifications with a call to the static member function `os_schema_evolution::set_evolved_schema_db_name()` before calling `evolve()`. This function takes a `const char*` as argument, the pathname of a compilation or application schema database (the compilation or application schema database for some other application).

Removed Classes

You must also specify the classes that are to be removed from the schema, that is, the classes present in the old schema but not in the new schema. (Removing a class from a schema results in deletion of all of its instances.) You do this with one call to the static member function `os_schema_evolution::augment_classes_to_be_removed()` for each removed class. This function is declared as follows:

```
static void augment_classes_to_be_removed(
  const char *name_of_class_to_be_removed
);
```

The calls should precede the call to `evolve()`.

You can also call this function once for all the classes to be removed if you pass an `os_Collection<char*>` containing the names of all the classes to be removed. In this case, you use the overloading

```
static void augment_classes_to_be_removed(
  const os_Collection<char*>
      &names_of_classes_to_be_removed
);
```

Again, this call should precede the call to `evolve()`.

Note that when you remove a class, `C`, you must also remove or modify any class that mentions `C` in its definition. Otherwise `err_se_cannot_delete_class` is signaled.

Work Database

In addition, you specify, also as an argument to `evolve()`, the pathname of the *work database* — a database to be created by the schema evolution facility and used internally as a scratch pad. This database holds the intermediate results of the

evolution process, allowing it to be restartable in case of interruption (due to network or system failure, or due to detection of an untranslatable pointer; see Untranslatable Pointers on page 220).

When evolution is interrupted, the work database records a consistent intermediate state of the evolution process. Subsequently calling `evolve()` by using the same work database causes evolution to be resumed from the point of interruption.

After evolution completes successfully, you should delete the work database.

Example: Changing the Type of a Data Member, Assignment-compatible Types

Consider an example that involves changing the value type of a data member where the types are assignment-compatible.

Suppose the schema for the database `/example/partsdb` starts out with the following definition of the class `part`:

Existing part class definition

```
class part {
  public:
    short part_id;
    part(short id) { part_id = id; }
    static os_typespec *get_os_typespec();
}
```

And you want to change the definition to be as follows:

New part class definition

```
class part {
  public:
    long part_id;
    part(long id) { part_id = id; }
    static os_typespec *get_os_typespec();
}
```

The value type of the data member `part_id` has changed from `short` to `long`. The constructor's argument type has also changed. Because C++ provides a standard conversion from `short` to `long`, migrated instances of the class `part` have their `part_id` fields initialized by assignment from the value of `part_id` in the corresponding old unmigrated instance.

Example: schema evolution application program

The application program that invokes the evolution process might look like the following:

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

#include "part1new.hh" /* the new definition */

main() {
  os_schema_evolution::initialize(0,0,0);
  os_schema_evolution::evolve(
    "/example/workdb",
```

```
    "/example/partsdb"
  os_schema_evolution::shutdown();
  );
}
```

Note that the header file `ostore/schmevol.hh` is included.

The argument `/example/workdb` is a name for the scratch pad database and the argument `/example/partsdb` specifies the database to be evolved.

Example: evolution program for multiple databases

An application that evolves several databases might look like the following:

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

#include "part1new.hh" /* the new definition */

main() {
  os_schema_evolution::initialize(0,0,0);
  os_Collection<char*> the_dbs_to_evolve;

  the_dbs_to_evolve |= "/example/partsdb1";
  the_dbs_to_evolve |= "/example/partsdb2";
  the_dbs_to_evolve |= "/example/partsdb3";

  os_schema_evolution::evolve(
    "/example/workdb",
    the_dbs_to_evolve
  );
  os_schema_evolution::shutdown();
}
```

Note that both versions of the `main()` program include the new definition of the modified class. The schema source file for this executable should also contain the new definition of the class `part`.

Schema source file with new definition of part class

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

#include "part1new.hh" /* this contains the new definition */

OS_MARK_SCHEMA_TYPE(part);
```

The instance migration phase of the schema evolution process migrates the parts in `/example/partsdb` (for the first version of `main()`), changing the size of the `part_id` field from the size of an `int` to the size of a `long`. As mentioned, the instance migration process also initializes the field by assignment from the preevolution value. This happens for all instances of the class `part`.

Note that the constructor for the new version of the class has no bearing on the initialization of migrated instances. The existing instances of the modified class are initialized according to the rules of default initialization described in these paragraphs. The new constructor initializes only those instances of the class that are created after evolution has occurred.

Using ossevol for Simple Schema Evolution

For a simple evolution such as this one, one that involves no transformers or user-defined handler functions, you can also use the ObjectStore utility `ossevol` instead of an application program. The utility takes arguments for the pathname of a work database, the pathname of a compilation or application schema database specifying the new schema, and the pathnames of the databases to evolve. For example:

```
ossevol /example/workdb /example/part.adb /example/partsdb
```

For information on the `ossevol` utility, see Using ossevol in Chapter 9, of the *C++ API User Guide*.

Using Transformer Functions

The instance initialization phase leaves migrated instances in a well-defined state. However, if you want to perform further application-specific processing on these instances as part of the migration process, you can supply transformer functions to accomplish this.

To do this, you define a transformer function for each class whose instances are to be transformed, then you associate the function with the class on whose instances the function will operate (see Associating a Transformer with a Class on page 209).

As part of the instance migration process, the ObjectStore schema evolution facility invokes each transformer function on each instance of its associated class. This includes each instance that is embedded in some other object, either as the value of a data member or as the subobject corresponding to a base class of the object's class.

The order of execution of transformers on embedded objects follows the same pattern as that of constructors. When the transformer for a given class is invoked, the transformers for base classes of the given class are executed first (in declaration order), followed by the transformers for class-valued members of the given class (in declaration order), after which the transformer for the given class itself is executed.

Signature of Transformer Functions

Transformers are functions with no return value and one argument of type `void*`. This argument is a pointer to the object being transformed, an instance of the new class.

Form of the call

```
void my_transform_function(void *the_new_obj)
```

Pre-evol transformer functions modify objects before any objects are moved or evolved; post-evol transformer functions modify objects after all objects are moved and evolved.

Associating a Transformer with a Class

With the transform function defined, you can associate the function with a class and invoke the evolution process. You make the association by calling the static member functions `os_schema_evolution::augment_pre_evol_transformers()` or `os_`

schema_evolution::augment_post_evol_transformers() in the application performing evolution. The call should be made before the call to os_schema_evolution::evolve().

The function augment_pre_evol_transformers() has the following two overloadings:

```
static void
  os_schema_evolution::augment_pre_evol_transformers(
    const os_transformer_binding&
  );

static void
  os_schema_evolution::augment_pre_evol_transformers(
    const os_Collection<os_transformer_binding*>&
  );
```

The function augment_post_evol_transformers() also has two overloadings:

```
static void
  os_schema_evolution::augment_post_evol_transformers(
    const os_transformer_binding&
  );

static void
  os_schema_evolution::augment_post_evol_transformers(
    const os_Collection<os_transformer_binding*>&
  );
```

You can construct an instance of os_transformer_binding by supplying a class name and a function pointer as arguments to the constructor:

```
os_transformer_binding("part", part_transform)
```

A typical call to augment_post_evol_transformers() would be

```
os_schema_evolution::augment_post_evol_transformers (
  os_transformer_binding("part", part_transform)
);
```

Example: Changing the Type of a Data Member, Assignment-incompatible Types

Now consider an example that changes the type of a data member to a type that is not assignment-compatible.

Suppose that you want to change the value type of the data member part_id of the class part from int to char* so that arbitrary strings can be used for part IDs.

There is no standard C++ conversion from int to char*. For schema evolution, a three-part sequence is used to add the new data member and copy the value from the old data member.

1 Perform a schema evolution that adds the new data member and retains the old data member.

2 Use a standard ObjectStore application (not a schema evolution application) to convert and copy the data from the old data member to the new one.

3 Perform a second schema evolution to remove the old data member.

Here are the relevant class definitions:

Existing part class definition

```
class part
{
public:
  int part_id;
  part(int id) { part_id = id; }
  static os_typespec *get_os_typespec();
}
```

Intermediate part class definition

For this type of schema evolution you need an intermediate schema that contains both the new data member as well as the old data member:

```
class part
{
public:
  int part_id;            // old data member
  char * part_id_string;  // new data member to replace it
   static os_typespec *get_os_typespec();
}
```

New part class definition

This is the class definition that includes the new data member, but not the old data member:

```
class part
{
public:
  char * part_id_string;
  part(char * id) {
    int len = strlen(id) + 1;
    part_id_string = new(os_cluster::with(this),
      os_typespec::get_char(), len) char[len];
    strcpy(part_id_string, id);
  }
  static os_typespec *get_os_typespec();
}
```

Here is a standard ObjectStore application that converts and copies the data from the old data member to the new data member. It does so by iterating through the database reading the contents of the old data member `part_id` and copying that information to the new data member `part_id_string`.

```
#include <stdio.h>
#include <ostore/ostore.hh>
#include <ostore/coll.hh>
#include "part2int.hh"
#include "dbname2.h"

static void part_transform(part* part_ptr)
{
  /* get the old data member value */
  int the_old_val = part_ptr->part_id;

  /* convert the old value to string form */
  char conv_buf[16];
```

```
  sprintf(conv_buf, "%d", the_old_val);

  int len = strlen(conv_buf) + 1;

  /* store it into the new data member */
  part_ptr->part_id_string =
    new(os_cluster::with(part_ptr), os_typespec::get_char(),
    len) char[len];
  strcpy(part_ptr->part_id_string, conv_buf);
}

int main(int, char **)
{
  objectstore::initialize();
  os_collection::initialize();
  OS_ESTABLISH_FAULT_HANDLER

  printf("evolve2: opening database %s\n", example_db2_name);
  os_database *db = os_database::open(example_db2_name);

  printf("evolve2: beginning a transaction\n");
  OS_BEGIN_TXN(txn,0,os_transaction::update)

  printf("evolve2: finding parts database root\n");
  os_database_root * parts_root =
    os_database_root::find("parts", db);
  os_typespec parts_typespec("os_Collection<part*>");

  printf("evolve2: walking through parts collection\n");
  os_Collection<part*> &parts =
    *(os_Collection<part*>*)
    parts_root->get_value(&parts_typespec);
  os_Cursor<part*> cursor(parts);
  for (
    part * p = cursor.first(); cursor.more(); p = cursor.next())
  part_transform(p);

  printf("evolve2: ending transaction\n");
  OS_END_TXN(txn)

  printf("evolve2: closing database %s\n\n", example_db2_name);
  db->close();

  OS_END_FAULT_HANDLER

  objectstore::shutdown();

  return 0;
}
```

Example: Changing Inheritance

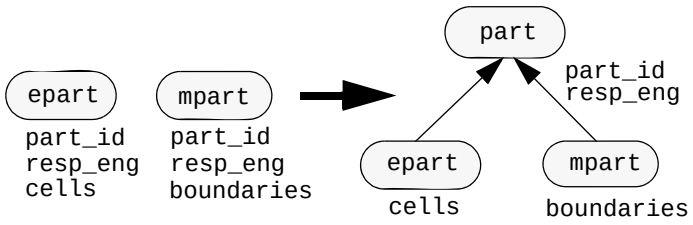
Following is an example that involves deleting some data members from a class and changing the class to inherit from a new base class.

Consider a database schema that uses the classes `epart`, for electrical part, and `mpart`, for mechanical part, and suppose that both these classes have data members for `part_id` and `responsible_engineer`. The following example shows the way to add a common base class, `part`, to these two classes, and how to move the common data members out of the definitions of `epart` and `mpart` and into the definition of `part`.

This schema change involves redefining `epart` and `mpart` by

- Deleting the members `epart::part_id`, `epart::responsible_engineer`, `mpart::part_id`, and `mpart::responsible_engineer`
- Making the classes inherit from the new class `part`, which has members `part::part_id` and `part::responsible_engineer`

Changing epart and mpart to inherit from part



Note that the schema evolution facility does not view the old member `epart::part_id` as related to the new member `part::part_id` (and similarly for `part::responsible_engineer`). It would be undesirable for the facility to make any assumptions about the semantic relationship between the two members based only on sameness of name, because this is an application-dependent matter.

Consequently, moving a data member from subtype to supertype should be viewed as deletion of the data member from the subtype together with addition of a new, distinct data member to the supertype. Similar remarks apply for moving members the other way, from supertype to subtype.

This schema evolution process takes two schema evolutions;. In order to copy data from the old members `epart::part_id` and `epart::responsible_engineer` to the new members `part::part_id` and `part::responsible_engineer` (and the similar data members in `mpart`), the schema evolution process needs an intermediate schema that contains both the old and new data members. In this example, copying the data is done in the first schema evolution process using the `augment_post_evol_transformers()` function. The second evolution is accomplished with the `ossevol` utility; this operation deletes the subtype data members that are no longer needed.

Following are the old, intermediate and new class definitions:

Old epart class definition

```
class epart {
  public:
    int part_id;
    employee *responsible_engineer;
    os_Collection<cell*> cells;
    . . .
```

```
    epart(int id, employee *eng) {
      part_id = i;
      responsible_engineer = eng;
    }
}
```

Old mpart class definition

```
class mpart {
  public:
    int part_id;
    employee *responsible_engineer;
    os_Collection<brep*> boundaries;
    . . .
    mpart(int id, employee *eng) {
      part_id = i;
      responsible_engineer = eng;
    }
}
```

Intermediate part class definition

```
class part
{
  public:
    int part_id;
    employee * responsible_engineer;
    part(int id, employee * eng) {
      part_id = id;
      responsible_engineer = eng;
    }
}
```

Intermediate epart class definition

```
class epart : public part
{
  public:
    /* next two fields will be deleted in the final schema */
    int part_id;
    employee * responsible_engineer;
    os_Collection<cell*> cells;
    epart(int id, employee * eng) : part(id, eng) {}
    static os_typespec *get_os_typespec();
}
```

Intermediate mpart class definition

```
class mpart : public part
{
  public:
    /* next two fields will be deleted in the final schema */
     int part_id;
     employee * responsible_engineer;
     os_Collection<brep*> boundaries;
    mpart(int id, employee * eng) : part(id, eng) {}
    static os_typespec *get_os_typespec();
}
```

New part class definition

```
class part {
  public:
    int part_id;
    employee *responsible_engineer;
    part(int id, employee *eng) {
      part_id = i;
      responsible_engineer = eng;
    }
}
```

New epart class definition

```
class epart : public part {
  public:
    os_Collection<cell*> cells;
    . . .
    epart(int id, employee *eng) : part(id, eng) {}
}
```

New mpart class definition

```
class mpart : public part {
  public:
    os_Collection<brep*> boundaries;
    . . .
    mpart(int id, employee *eng) : part(id, eng) {}
}
```

New schema source file

The schema source file for this executable should contain the new definitions of `epart` and `mpart`, as well as the definition of `part`.

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

#include "part3int.hh"

OS_MARK_SCHEMA_TYPE(employee);
OS_MARK_SCHEMA_TYPE(epart);
OS_MARK_SCHEMA_TYPE(mpart);
```

The instance migration phase of the schema evolution process modifies the instances of `epart` and `mpart` by adding to each instance a subobject corresponding to the base class and initializes it as if by a constructor that initializes each member to `0`.

Supplying a transformer function for each derived class

In order to overwrite the default initialization performed by the schema evolution facility and initialize `part::part_id` and `part::responsible_engineer` for a migrated instance based on the values of the old `part_id` and `responsible_engineer` fields for the corresponding unmigrated instance, you supply a transformer function for each derived class — `epart` and `mpart`.

```
static void epart_transform(os_void_p the_new_obj)
{
  /* get the old data member values from the derived class */
  epart * epart_ptr = (epart *)the_new_obj;
  int the_old_id_val = epart_ptr->part_id;
  employee* the_old_resp_eng_val =
      epart_ptr->responsible_engineer;

  /* set the new data member values in the base class */
  part* part_ptr = epart_ptr;
  part_ptr->part_id = the_old_id_val;
  part_ptr->responsible_engineer = the_old_resp_eng_val;
}

static void mpart_transform(os_void_p the_new_obj)
{
  /* get the old data member values from the derived class */
  mpart * mpart_ptr = (mpart *)the_new_obj;
  int the_old_id_val = mpart_ptr->part_id;
  employee* the_old_resp_eng_val =
      mpart_ptr->responsible_engineer;

  /* set the new data member values in the base class */
```

```
  part * part_ptr = mpart_ptr;
  part_ptr->part_id = the_old_id_val;
  part_ptr->responsible_engineer = the_old_resp_eng_val;
}
```

The transformer functions for the two classes need to do essentially the same thing. Each function retrieves the old values for `part_id` and `responsible_engineer` in the derived class and sets the new values for `part::part_id` and `part::responsible_engineer` accordingly.

If the current evolution calls for the migration of instances of the class `employee`, the value of `responsible_engineer` retrieved from the old instance is a pointer to the new employee instance corresponding to the original data member value. This is because pointers to migrated objects are modified during the initialization phase to point to the new instances. This turns out to be convenient because you are usually interested in the evolved version of the old data member value.

Example: associating transformers with their classes and invoking evolution

Following is an application that associates the transformers with their classes and invokes evolution:

```
int main(int, char **)
{
  os_schema_evolution::initialize(0,0,0);

  /* associate epart_transform with the class epart */
  os_schema_evolution::augment_post_evol_transformers(
      os_transformer_binding("epart", epart_transform) );

  /* associate mpart_transform with the class mpart */
  os_schema_evolution::augment_post_evol_transformers(
      os_transformer_binding("mpart", mpart_transform) );

  /* perform the evolution process */
  os_schema_evolution::evolve(
      example_wdb3_name,example_db3_name);

  os_schema_evolution::shutdown();

  return 0;
}
```

Finally, to remove the old data members you use the `ossg` utility to generate a new application schema using the new class definition and then run the `ossevol` utility. For more information on `ossg` and `ossevol`, see Chapter 4, Utilities in *Managing ObjectStore*.

For databases undergoing the evolution described in this example, ObjectStore detects as untranslatable any pointers to `eparts` or `mparts` typed as `void*`. This is because, for example, before evolution, such a pointer to an `epart` could also be interpreted as referring to the value of `epart::part_id` (because this `int` object starts at the same point as the `epart`), while after evolution it could no longer be interpreted as referring to that object. For more information on untranslatable pointers, see Untranslatable Pointers on page 220.

If the example is modified to include a leftmost base class for `epart` and `mpart`, both before and after evolution, `void*` pointers to `eparts` and `mparts` are not untranslatable.

Instance Reclassification

The ObjectStore schema evolution facility lets you migrate an instance to a subclass of its original class. You can do this only when the original class is not a virtual base class of the new class. Reclassifying an instance is particularly useful when you are adding a derived class to a schema and this derived class is a more appropriate class than the base class for existing instances.

To reclassify an instance:

1 Define the new derived class so that it

- Does not define nonstatic data members. This is a temporary restriction. You can define nonstatic data members in a later step in this procedure.
- Does not define virtual functions. This is a temporary restriction.
- Has the instance's original class as its base class.
- Has no other base classes. This is a temporary restriction.

These restrictions ensure that the initial storage layout of the derived class is the same as the storage layout of the original class. In a later reclassification step, you can change the storage layout.

2 Write and execute a program that

a Visits in the database each object that you want to reclassify

b Applies your application-specific criteria to determine how to reclassify the instance

c Calls `objectstore::change_type()` to reclassify the instance. See *C++ API Reference*, `objectstore::change_type()`.

This program uses a modified application schema that contains the new derived class. You create this application schema as you would any other application schema.

After this step, each instance is an instance of the new desired class, but you have not changed the storage layout.

3 Redefine the new derived class so that it has the data members you want. You can add virtual functions and additional base classes, virtual or not, to the definition. Do not change the definition of the base class yet.

4 Run the `ossg` utility to generate an application schema that contains the new derived class as well as the other classes in your application.

5 Write and execute a schema evolution program that evolves the database to this new application schema. Your schema evolution program must define a post-evolution transformer function that sets the nonstatic data members of the derived class according to the appropriate data members of the base class.

6 Redefine the base class to remove the data members that have been replaced by data members of the new derived class.

7 Run the `ossg` utility to generate an application schema that contains the new base class, the new derived class, and any other classes in your application.

8 Run the `ossevol` utility to evolve the database to this new and final application schema.

9 Modify your application program to use the new and final application schema.

10 Build and test your program.

Example of Reclassifying Instances

For example, consider a schema that contains a definition for the `part` class. The `part` class defines two data members:

- `cells` is a pointer to a collection of subcircuits of an electrical part.
- `boundary_rep` is a pointer to a geometric representation of the boundary of a mechanical part.

When a `part` instance has a nonnull, nonzero value for the `cells` data member, it has a value of `0` for the `boundary_rep` data member. Conversely, when a `part` instance has a nonnull, nonzero value for the `boundary_rep` data member, it has a value of `0` for the `cells` data member.

You want to modify this schema to include two new classes that are derived from the `part` class:

- The `epart` (electrical part) class will contain the `cells` data member.
- The `mpart` (mechanical part) class will contain the `boundary_rep` data member.

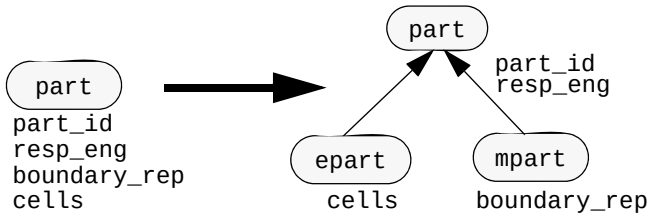
The `part` class will no longer define the `cells` and `boundary_rep` data members. In addition to adding the subclasses to the schema, you need to migrate existing instances of the `part` class so that

- Instances that have a nonnull, nonzero value for `cells` are reclassified as instances of the `epart` class.
- Instances that have a nonnull, nonzero value for `boundary_rep` are reclassified as instances of the `mpart` class.

To summarize, the schema changes you want to maker are:

- Delete the `cells` and `boundary_rep` data members from the `part` class.
- Derive the `epart` and `mpart` classes from the `part` class.

The following figure illustrates this:



Sample code that does this is Example 4 in the `examples/sevol` directory of your ObjectStore installation directory. The following files make up this example. On Windows. the source files have a `.cpp` extension. On other platforms, the source files have a `.cc` extension.

`db_pop4`	Creates the sample database.
`part4pre.hh`	Defines the schema that the `db_pop` program uses.
`part4int.hh`	Defines the first intermediate version of the modified schema. That is, it defines the two new derived classes, `epart` and `mpart`, but the class definitions do not include nonstatic data members.
`chtype4`	Visits each `part` instance in the database and invokes the `objectstore::change_type()` function to reclassify each instance as an instance of `epart` or `mpart` according to logic in the `chtype4` program. This program uses a collection cursor to find all `part` objects. If the database did not contain a collection of `part` objects, a program could use an `os_dynamic_extent` cursor.
`part4int2.hh`	Defines the second intermediate version of the modified schema. That is, it defines the two new derived classes, `epart` and `mpart`, so that the definitions include all desired data members, including nonstatic data members. The definition of the `part` class, which is the base class for the two new derived classes, remains unchanged.
`evolve4`	Evolves the database to the application schema generated with the `part4int2.hh` schema source file.
`part4new.hh`	Defines the final schema of the evolved database. That is, the `part` class no longer contains the `cells` and `boundary_rep` data members. At this point, you run the `ossevol` utility: `ossevol wdb4.db sch4anew.adb db4.db` Then you would modify your application to use part4new.hh, and you would build and test your program.
`db_read4`	Reads the database and displays information about the objects it finds.

Untranslatable Pointers

During the instance initialization phase of schema evolution, ObjectStore adjusts all pointers and references to instances of modified classes so that they point to the new, migrated instances of these classes. During this process, ObjectStore might detect various kinds of untranslatable pointers or references. For example, it might detect a pointer to the value of a data member that has been removed in the new schema. By default, an exception is signaled when an untranslatable pointer or reference is encountered.

Using Handler Objects

Alternatively, you can provide a handler object to handle one or more of the following categories of untranslatable pointers and references:

- Untranslatable pointers and C++ references to objects
- Untranslatable ObjectStore references
- Untranslatable pointers and C++ references to members
- Untranslatable database root values

Each time an untranslatable pointer or reference is detected, the untranslatable pointer object handles it and then schema evolution is resumed. A handler object cannot modify any data in the databases being evolved; however, the object can return a value that causes the pointer to be retranslated, either to a different path or to null. The object can also generate text output. For example, you can record the location of an untranslatable pointer by creating a transient ObjectStore reference to the untranslatable pointer and then dumping its text representation to a file (see `os_reference_internal::dump()` in Chapter 2 of the *C++ API Reference*). This text representation can be used by a subsequent process to create another ObjectStore reference to the same untranslatable pointer (see `os_Reference<T>::os_Reference()` in Chapter 2 of the *C++ API Reference*).

When you invoke the untranslatable pointer handler for a pointer-to-member type, you should not call the following functions:

- `os_untranslatable_pointer_handler::get_target_segment()`
- `os_untranslatable_pointer_handler::get_target_cluster()`
- `os_untranslatable_pointer_handler::get_target_offset()`

These functions either do not return valid values or they fail. This is because a pointer-to-member type does not point to an instance of the owner class. To detect if the handler was invoked for a pointer-to-member type, call the `os_untranslatable_pointer_handler::isPTOM()` function.

Creating a Handler Object

To create a handler object:

- Derive a class from `os_untranslatable_pointer_handler`.
- Override the class's `handle_`*`xxx`* functions that are appropriate.

- Override the class's `clone()` function.
- Register the class with a call to the static member function `os_schema_evolution::set_untranslatable_pointer_handler()`.

The `handle_xxx` virtual functions you can override are as follows:

- `handle_ambiguous_void_pointer()`
- `handle_deleted_sub_object()`
- `handle_evolved_to_bit_field()`
- `handle_evolved_to_incompatible_type()`
- `handle_evolved_to_smaller()`
- `handle_evolved_to_static()`
- `handle_wrong_type_pointer()`

For more information on these functions, see `os_untranslatable_pointer_handler` in the *C++ API Reference*.

Understanding Untranslatable Pointers

Besides this categorization, there is another, orthogonal way of categorizing untranslatable pointers and references. This categorization will help you understand the pointers and references that are counted as untranslatable.

Typed pointers and references to deleted subobjects

The instance migration process deletes subobjects of instances of a given class when either

- The subobject is the value of a data member that has been removed from the class.
- The subobject corresponds to a class that the given class previously inherited from, but no longer does.

Any pointer or reference to such a deleted subobject is untranslatable and can signal the exception `err_se_deleted_object` or `err_se_deleted_component`.

void* pointers and collocation ambiguities

A `void*` pointer in an ObjectStore database has an associated set of objects, the objects collocated at the region of memory it points to. These are all the objects to which the pointer can be interpreted as referring, instances of the types to which the pointer can legitimately be cast.

For example, a `void*` pointer to an instance of the class `epart` from the preevolution schema of Example: Changing Inheritance on page 212 also points to the beginning of memory occupied by an `int`, the value of the member `epart::part_id`.

If a `void*` pointer is associated, before evolution, with an object with which it is not associated after evolution, the pointer is untranslatable and can signal the exception `err_se_ambiguous_void_pointer`.

Consider again Example: Changing Inheritance on page 212. After evolution, the `void*` pointer to an instance of `epart` now also points to a `part`, as well as an `int`, the value of the member `part::part_id`. However, while before evolution the pointer could be interpreted as referring to the value of `epart::part_id`, after evolution it no longer could be interpreted as referring to this object. Because the value of `epart::part_id` is no longer one of the pointer's associated objects, the

pointer becomes untranslatable. (Remember that ObjectStore makes no semantic connection between `epart::part_id` and `part::part_id`.)

Note that `void*` pointers appear in every database because the values of database roots are typed as `void*`. They might be common in some databases because in the underlying representations of ObjectStore collections, elements are typed as `void*`.

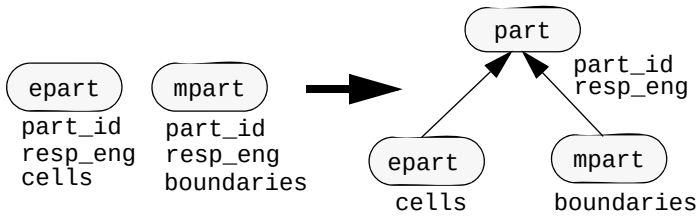
Pointers and references to transient or freed memory and type-mismatched pointers and references are untranslatable even before schema evolution; however, ObjectStore detects them during instance initialization. Pointers and references to transient objects or to objects that have been deleted are untranslatable. Pointers and references with particular types that are not actually the addresses of some objects of that type are also untranslatable.

Example: Using Untranslatable Pointer Handlers

Consider the schema change made in Example: Changing Inheritance on page 212.

Changing epart and mpart to inherit from part

Change `epart` and `mpart` to inherit from `part`, factoring out the common state to the base type.



As described previously, if a database undergoes this schema change and it contains `void*` pointers to `eparts` or `mparts`, these pointers are detected as untranslatable and should be handled with an untranslatable pointer handler.

A `void*` pointer to (for example) an `epart` is untranslatable because it could be interpreted, before evolution, as referring to the value of `epart::part_id`, which does not exist after evolution. However, if you know that this interpretation is never intended, you can use the following untranslatable pointer handler:

Example: using an untranslatable pointer handler

```
#include <ostore/ostore.hh>
#include <ostore/coll.hh>
#include <ostore/schmevol.hh>
#include <ostore/mop.hh>
#include <ostore/osmop/os_path.hh>
#include "part5int.hh"

class my_untranslatable_pointer_handler : public
    os_untranslatable_pointer_handler
{
public:
  virtual os_path_to_data* handle_ambiguous_void_pointer()
  {
```

```
    char* path_string = get_target_path().name();

    if (strcmp(path_string, "epart.part_id") == 0
        || strcmp(path_string, "mpart.part_id") == 0) {
      /**
       ** We know that these void * pointers in the pre
       ** evolved world should be void * pointers to parts
       ** in the post evolved world so we set the pointer
       ** to the evolved object.
       **/
      os_path_to_data& new_target_path =
          get_target_path().outer_path();
      new_target_path.pop();
      /* now new_target_path points to epart or mpart */
      delete path_string;
      return &new_target_path;
    } else {
      /* unanticipated ambiguous pointer, signal exception */
      delete path_string;
      return _default_handler();
    }
  }
  virtual os_untranslatable_pointer_handler* clone()
  {
    return new my_untranslatable_pointer_handler(*this);
  }
}
```

Using transformers with untranslatable pointer handlers

This example uses the same transformers as the example in Example: Changing the Type of a Data Member, Assignment-incompatible Types on page 210. Following is an application that associates the transformers with their classes, registers the untranslatable pointer handler, and invokes evolution:

```
#include <ostore/ostore.hh>
#include <ostore/coll.hh>
#include <ostore/schmevol.hh>
#include <ostore/mop.hh>
#include <ostore/osmop/os_path.hh>
#include "part5int.hh"

int main(int, char **)
{
  os_schema_evolution::initialize(0,0,0);

  /* register the ambiguous pointer handler */
  os_schema_evolution::set_untranslatable_pointer_handler(
    new my_untranslatable_pointer_handler
  );

  /* associate epart_transform with the class epart */
  os_schema_evolution::augment_post_evol_transformers(
    os_transformer_binding("epart", epart_transform)
  );

  /* associate mpart_transform with the class mpart */
  os_schema_evolution::augment_post_evol_transformers(
    os_transformer_binding("mpart", mpart_transform)
  );
```

```
  /* perform the evolution process */
  os_schema_evolution::evolve(
    example_wdb5_name, example_db5_name
  );
  os_schema_evolution::shutdown();
  return 0;
}
```

Handler method that changes pointers

The following example recognizes pointers to certain data members and changes one data member to a null pointer and another data member to a pointer to a different data member. It shows the use of one handler method.

```
virtual os_path_to_data* handle_deleted_sub_object() {
  char* source_path_string = get_source_path().name();

  if (strcmp(source_path_string,
          "ThePointers.nulled.deleted_pointer") == 0) {
    /* Change this invalid pointer to a null pointer */

    delete source_path_string;
    return NULL;
  } else if (strcmp(source_path_string,
          "ThePointers.retained.deleted_pointer") == 0) {
      /* Change this invalid pointer to a valid one - substitute
         new_target for deleted_target */

      os_path_to_data* new_path =
        os_path_to_data::make(get_target_path());
      const os_class_type& the_class = new_path->get_root();
      new_path->pop();
      new_path->add(*the_class.find_member_variable("new_target"));

      delete source_path_string;
      return new_path;
  } else {
    /* Unexpected, call the default handler */

    delete source_path_string;
    return _default_handler();
  }
}
```

Obsolete Index and Query Handlers

When the selection criterion of a query or the path of an index makes reference to a removed class or data member, or makes incorrect type assumptions in light of a schema change, the query or index becomes obsolete. ObjectStore detects all obsolete queries and indexes. As with untranslatable pointers, you can handle obsolete queries or indexes by providing special handler functions for each purpose. If you do not supply handlers, ObjectStore signals an exception when it detects an obsolete query or index. If you supply an obsolete query handler function that allows the schema evolution to continue when it encounters an obsolete query, ObjectStore internally marks the query so that subsequent attempts to use it result in the exception `err_os_query_evaluation_error`.

To handle obsolete queries or indexes,

- Define a function with the appropriate signature.
- Register the function with a call to the static member function `os_schema_evolution::set_obsolete_index_handler()` or `os_schema_evolution::set_obsolete_query_handler()`.

The signature for an obsolete query handler is

```
void function_name(os_coll_query &query,
    const char *query_expr)
```

A reference to the obsolete query is passed in together with a string expressing the query's selection criterion.

The signature for an obsolete index handler is

```
void function_name(os_collection &coll,
    const char *path_string);
```

A reference to the collection indexed by the obsolete index is passed in with a string expressing the index's path (key).

Task List Reporting

Before initiating evolution for a particular schema change, you might want to generate a task list to verify your expectations concerning the instance initialization phase. The task list contains a function definition for each class whose instances are migrated.

Form of the call

Each function has a name of the form

class-name`@[1]::initializer()`

where *class-name* names the function's associated class.

Statements for data members and their classes

Each function definition contains a statement or comment for each data member of its associated class. For a member with value type `T`, this statement or comment is any of

- Assignment statement
- Call to `T@[1]::copy_initializer()`
- Call to `T@[2]::construct_initializer()`
- Call to `T@[1]::initializer()`
- Comment indicating that the field is initialized to `0`

Assignment statements

An assignment statement is used when the old and new value types of the member are assignment compatible:

- `T@[1]::copy_initializer()` is used when the member has not been modified by the schema change and the new value can be copied bit by bit from the old value.
- `T@[2]::construct_initializer()` is used when the value type has been modified and the new value type is a class.

- `T@[1]::initializer()` is used when the member has not been modified by the schema change, but instances of the value type of the member are migrated. Definitions for all these functions appear in the task list.

A program to generate a task list is exactly like a program to perform evolution except that the static member function `os_schema_evolution::task_list()` is called instead of `os_schema_evolution::evolve()`.

task_list() function

The function `task_list()` has two overloadings analogous to the two overloadings of `evolve()`, declared as follows:

```
static void task_list(
  const char *workdb_name,
  const char *db_to_evolve
);

static void task_list(
  const char *workdb_name,
  const os_collection &dbs_to_evolve
);
```

Using task_list()

Before calling `task_list()`, you use `os_schema_evolution::set_task_list_file_name()` to specify the file to which the task list is to be sent. This function is declared as follows:

```
static void set_task_list_file_name(const char *file_name);
```

As with `evolve()`, the new schema is, by default, the schema of the application that calls `task_list()`; however, you can specify the new schema with `os_schema_evolution::set_evolved_schema_db_name()` before calling `task_list()`.

Also, as with `evolve()`, you must specify the classes that are to be removed from the schema with `os_schema_evolution::augment_classes_to_be_removed()`. The calls should precede the call to `task_list()`.

Instance Initialization Rules

This section starts with a description of the various categories of schema evolution. Following this discussion, the initialization rules for each category are described.

Kinds of schema modifications

The different kinds of schema modification can be divided into three broad (not entirely disjoint) categories:

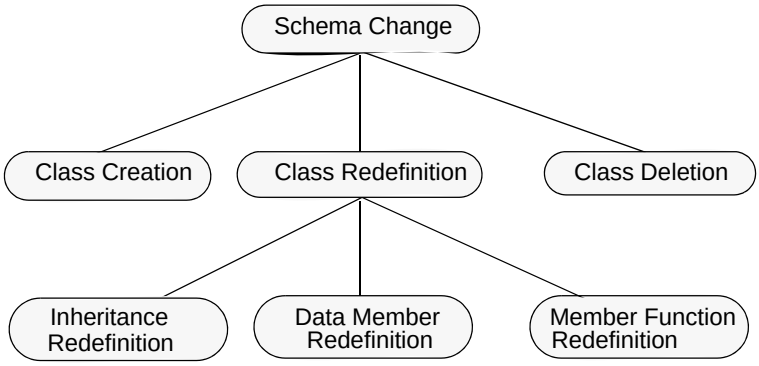
- Class creation
- Class redefinition
- Class deletion

Kinds of class redefinitions

The kinds of class redefinition, in turn, can be divided into three subcategories, changes relating to

- Inheritance
- Data members
- Member functions

The following diagram shows the categories and subcategories of schema modification:



Class Creation

Adding a class to a database's schema never, by itself, requires the use of the schema evolution facility. This is because a new class cannot have any previously existing instances. Because there cannot be any existing instances, instance migration is not necessary, and adding the class to the database's schema is handled automatically when an application using the new class opens the database or when it creates the first persistent instance of the new class, in incremental schema installation mode.

Inheritance Redefinition

However, although adding a class does not by itself require using the evolution facility, sometimes adding a class involves also redefining another existing class. This is the case when you add a new class as a base class of another existing class, for example. The definition of the existing class must be changed to specify inheritance from the new class. And the representation of instances of the derived class must be supplemented with a subobject corresponding to the new base class. Such schema changes fall under the category of inheritance redefinition.

In general, inheritance redefinition includes changing a class to inherit from a new or existing class, changing a class so that it no longer inherits from an existing class, or changing class inheritance from virtual to nonvirtual or the reverse.

Data Member Redefinition

Class redefinition relating to data members includes changing the definition of a class by adding or deleting members, changing the value type of a data member, and changing the order of data members. (To change the name of a data member, you delete it and then add a new one with the desired name.)

Member Function Redefinition

There are only two kinds of member function-related changes that require schema evolution: changing the definition of a class by adding the first virtual function and

changing the definition of a class by removing the only virtual function. These modifications require schema evolution because they change the representation of any instances of the modified class. Other changes related to member functions have no effect on the layout of class instances, and so they do not require schema evolution.

Class Deletion

In the case of class deletion, instance migration consists of the deletion of existing instances of the deleted classes. Any pointers typed as pointers to a deleted class are detected before instance initialization and signal an `err_schema_evolution` exception. Any `void*` pointer to an instance of a deleted class (or pointer to a subobject of such an instance) is detected as an untranslatable pointer.

As with class creation, deleting a class might at the same time involve changing the inheritance structure of some other class. This is the case, for example, when you delete a class that serves as a base class of another class that is to remain in the schema. The definition of the remaining class must be changed so that it no longer specifies inheritance from the deleted class. And the representation of the remaining class's instances must have the subobject corresponding to the base class removed. Such schema changes fall under the category of inheritance redefinition as well as class deletion.

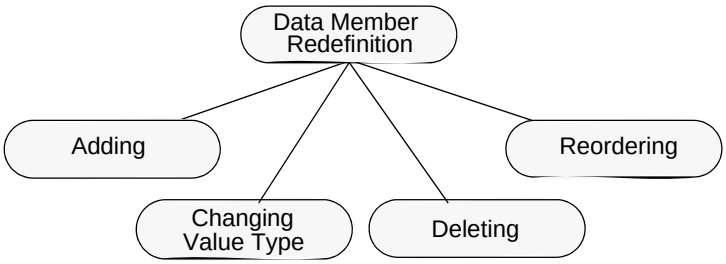
Schema Changes Related to Data Members

The sections that follow consider the different types of schema modification related to data members:

- Adding Data Members
- Deleting Data Members
- Changing the Value Type of a Data Member
- Changing the Order of Data Members
- Moving Data from One Member to Another
- Evolving Schemas That Contain Pointer-to-Member Types
- Summary of Data Member Changes Not Requiring Explicit Evolution

This section is particularly concerned with describing the instance migration phase of schema evolution for each kind of modification.

Categories of data member redefinition



Note that indirect instances of a modified class are migrated just as are direct instances. That is, if you change the definition of base class B, then instances of class D, derived from B, are migrated, just as are direct instances (if there are any) of B.

Adding Data Members

When you add a data member to a class, the schema evolution process changes the representation of any of its instances by adding a field to hold the value of the new member. The way this field is initialized depends on the value type of the new member.

If the value type is a built-in nonarray type (integral type, floating type, pointer type, reference type, enumeration type, or pointer-to-member type), it is initialized with the appropriate representation of `0`. If the value type is a class, the field is initialized as if by a constructor that initializes each member to `0`.

If the value type is an array type, each element of the array is initialized (for arrays of built-ins) with `0` or (for arrays of class instances) as if by a constructor that initializes each member to `0` for the array's element class. For arrays of arrays, these rules are applied recursively. In other words, an array is initialized by initializing each of its elements as if it were a separate data member.

As with all modified classes, the class with the new data member can have an associated transformer function that you supply. If you want, this function can overwrite these default initializations, supplying a value for the new field in whatever way meets your needs.

Deleting Data Members

When you delete a data member from a class, the schema evolution process changes the representation of any of its instances by removing the field that held the value of the deleted member. Because no new storage is created by this schema change, the issue of initialization does not arise.

By default, pointers to members being removed result in an untranslatable pointer exception during evolution. You can, however, supply an untranslatable pointer handler to process the untranslatable pointer and resume evolution. See Untranslatable Pointers on page 220.

Changing the Value Type of a Data Member

When you change the value type of a data member, the schema evolution process changes the representation of any of its instances by adjusting the size of the member's associated storage (if necessary) and reinitializing that storage. The way this storage is initialized depends on the new and old value types.

Consider first the case in which the new value type is not an array type.

Assignment-compatible value types

Old and new member declarations with assignment-compatible value types:

```
int cost;   /*old member declaration */
float cost;   /*new member declaration */
```

If the new and old types are assignment compatible, the new field is initialized by assignment. That is, ObjectStore assigns the value of the old data member to the storage associated with the new member, applying any standard conversions defined by the C++ language.

For example, if you change the value type of a data member from `int` to `float`, an old instance with the value `(int)(17)` for this member is changed to have value `(float)(17.0)`.

In some cases, schema evolution considers types assignment compatible when C++ would not. For example, if D is derived from B, schema evolution assigns a B* to a D* if it knows that the B is also an instance of D.

If the new and old types are not assignment compatible, there are two cases.

New value type is a built-in

Old and new member declarations with assignment-incompatible value types, where the new value type is a built-in:

```
class dollars cost;   /*old member declaration */
float cost;   /*new member declaration */
```

If the new value type is a built-in nonarray type (integer type, floating type, pointer type, reference type, enumeration type, or pointer-to-member type), it is initialized with `0`.

New value type is a class

Old and new member declarations, where the new value type is a class:

```
int cost;   /*old member declaration */
class dollars cost;   /*new member declaration */
```

If the new value type is a class, the field is initialized as if by a constructor that initializes each member to `0`.

If you change the value type of a data member by changing it from a `signed` integer type to an `unsigned` integer type or the reverse, you need not perform schema evolution. This is because such a change does not change the size of the associated field and does not change the way (sufficiently small) positive numbers are represented.

Array values with compatible types

Consider the case in which the new value type *is* an array type.

Old and new member declarations with array value types whose elements are assignment compatible:

```
int cost[10];   /*old member declaration */
float cost[10];   /*new member declaration */
```

If the old value type is also an array type, and if the element types of the arrays are assignment compatible, the new field is initialized by assignment. That is, ObjectStore assigns the value of the i^{th} element of the old array to the i^{th} element of the new array, applying any standard conversions defined by the C++ language. This is done for all *i* between 0 and one less than the size of the smaller array.

If the new array has *n* more elements than the old array, the trailing *n* elements of the new array are initialized with `0` (if the element type is a built-in nonarray type) or as if by a generated default constructor (if the element type is a class). If the old array has *n* more elements than the new array, the trailing *n* elements of the old array are ignored.

Array values with incompatible types

Old and new member declarations with array value types whose elements are not assignment compatible:

```
class dollars cost[10];   /*old member declaration */
float cost[10];   /*new member declaration */
```

If the old value type is also an array type but the element types are not assignment compatible, each element of the new array is initialized with `0` (if the element type is a built-in nonarray type) or as if by a constructor that initializes each member to `0` (if the element type is a class).

Nonarray type to array type

Old and new member declarations; the old value type is a nonarray type and the new value type is an array type.

```
int cost;   /*old member declaration */
float cost[10];   /*new member declaration */
```

If the old value type is not an array type, each element of the new array is initialized with `0` (if the element type is a built-in nonarray type) or as if by a constructor that initializes each member to `0` (if the element type is a class).

In general, you initialize arrays by initializing each array element as if it were a separate data member.

For a multidimensional array, these rules apply to the first dimension and recursively to the other dimensions if the length of each other dimension is not changed by evolution. If the length of one of these other dimensions changes, every element of the multidimensional array is initialized with `0` (if the element type is a built-in) or as if by a constructor that initializes each member to 0 (if the element type is a class).

As with all modified classes, the class with the modified data member can have an associated transformer function that you supply. If you want, this function can overwrite these default initializations, supplying a value for the new field in whatever way meets your needs.

Bit fields are evolved according to the default signed/unsigned rules of the implementation that built the evolution application. This can lead to unexpected results when an evolution application built with one default rule evolves a database originally populated by an application built by an implementation whose default rule differs. The unexpected results occur when the evolution application attempts to increase the width of a bit field.

Changing the Order of Data Members

When you change the order of the data members defined by a class (by changing the order in which their declarations appear within the definition of the class), the schema evolution process changes the representation of any of its instances by reordering the storage fields associated with the members. Because there is no new storage created by this schema change, the issue of initialization does not arise.

Moving Information from One Data Member to Another

If the goal of your schema evolution is to move information from one data member to another, use the following procedure:

1. Do a schema evolution to add the new data member which will be initialized by default to zero. Do not remove the old data member in this step. You can use the `ossevol` utility for this step.
2. Run a user program that initializes the new data member according to the contents of the old data member. This program uses ordinary operations, not meta-object protocol. It finds all the instances of the evolved class in one of the following ways, the choice is whatever the user prefers:
   - Using an object cursor (the slowest but easiest way).
   - Navigating through application-specific data structures.
   - Incrementally, the first time the application program needs the value of the new data member. In this case, step 1 might also add a flag data member that is set nonzero after the new data member has been initialized.

     Zero in the flag means the incremental initialization must be called.
   - during the schema evolution in step 1, using an `augment_post_evol_transformer` function. Schema evolution will automatically call this function once on each instance of the evolved class. In this case step 1 and step 2 run in the same user program and step 1 cannot be done by the `ossevol` utility.
3. Run a cleanup schema evolution that removes the old data member and (if there is one) the flag data member. This cleanup can be done at any convenient time since it makes no logical difference, it only saves storage.

Evolving Schemas That Contain Pointer-to-Member Types

You can evolve a schema that contains pointer-to-member types.

However, support for evolving schemas that contain pointer-to-member-function types is restricted. When one of the following conditions is met, the schema evolution facility copies pointer-to-member-function types to the evolved database:

- You do not evolve the owner class, and you do not evolve the pointer-to-member-function type so that it points to a member of a base class instead of a member of a derived class. Likewise, you do not evolve it so that it points to a member of a derived class instead of a member of a base class.

  (The owner class is the class that contains the member that is the target of the pointer.)

- The value of the pointer-to-member-function type is null.

Summary of Data Member Changes Not Requiring Explicit Evolution

You need not invoke schema evolution to make the following kinds of data member modifications:

- Changing the value type of a data member from a signed type to `unsigned` type and the reverse
- Changing the access specified for a data member (`private`, `public`, or `protected`)
- Changing the value type of a data member from a `const` to non-`const` type and the reverse
- Adding or removing `static` data members

Schema Changes Related to Member Functions

As mentioned earlier, there are only two kinds of member-function-related changes that require schema evolution: changing the definition of a class by adding the first virtual function and changing the definition of a class by removing the only virtual function. These modifications require schema evolution because they change the representation of any instances of the modified class. Other changes related to member functions have no effect on the layout of class instances and, therefore, do not require schema evolution.

Schema Changes Related to Class Inheritance

Changes relating to class inheritance include adding base classes, removing base classes, and changing class inheritance from virtual to nonvirtual, or the reverse. Each of these is discussed in the following sections:

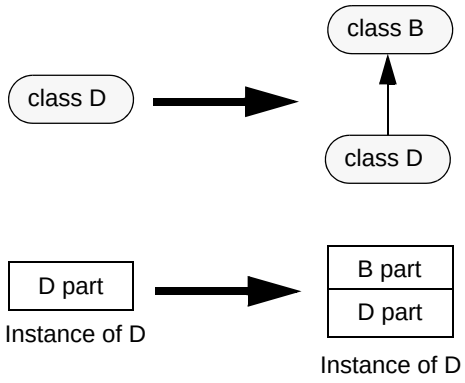
- Adding Base Classes
- Removing Base Classes

- Inserting Base Classes
- Changing Between Virtual and Nonvirtual Inheritance

Adding Base Classes

When you modify a database's schema by adding a base class (B in the following figure) to an existing class (D in the following figure), instances of D must be supplemented with a B part.

Adding a base class to an existing class



When the class D is modified to inherit from a base class, B, its instances must be modified to include a B part.

The instance initialization phase of schema evolution adds the B part to each instance of D and initializes that part as if by a constructor that initializes each member to `0`.

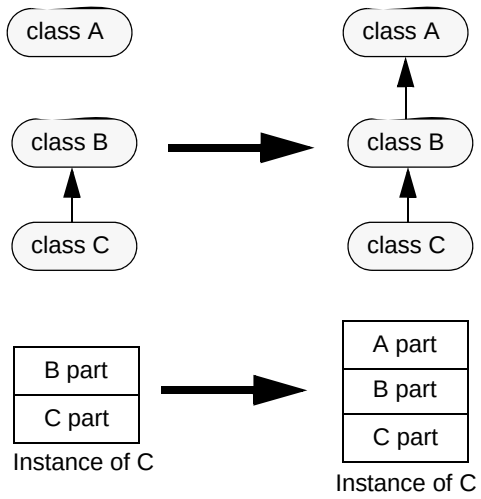
If you provide a transformer function for D, it is run during the instance transformation phase.

This category of schema change covers more cases than might be suggested by the previous illustration.

In particular,

- Schema evolution works the same if B is added as a base class to more than one existing class. Each instance of each existing class must be supplemented with a B part.
- Indirect as well as direct instances of a class made to inherit from a base class must be migrated. See When changing a class requires migrations on page 235.
- The class that is added as a base class might or might not be part of the old schema. In either case, no instance migration need be performed for the base class unless it too has evolved.

When changing a class requires migrations

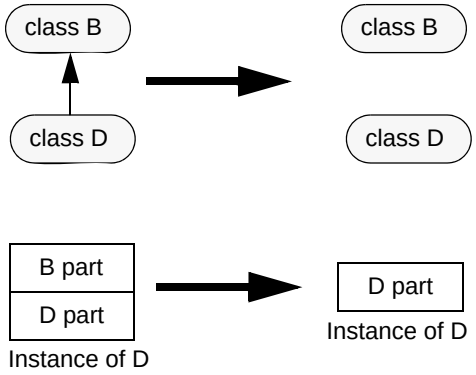


When you change the definition of B so that it inherits from A, instances of C (derived from B) must be migrated.

Removing Base Classes

When you change a class, D, so that it no longer inherits from a given class, B, each instance of D is migrated by removing the subobject corresponding to B.

Modifying other instances when removing a class

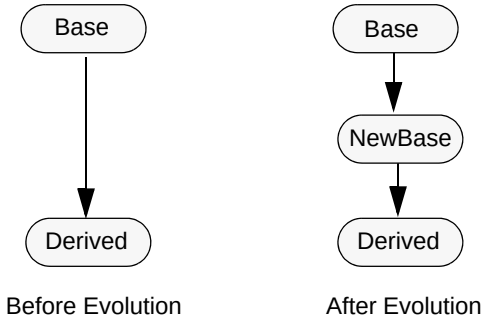


When the class D is modified so that it no longer inherits from a base class, B, its instances must be modified to remove the B part.

Pointers to the subobject being removed, if they are typed as B* rather than D*, result in the signaling of an untranslatable pointer exception during evolution. (Pointers typed as D* are, of course, adjusted automatically to point to the migrated instance of D.) The same is true for pointers (so typed) to data members of the deleted subobject.

Inserting Base Classes

Sometimes you want to remodularize your classes to introduce a new base class that stands between an existing base class and its derived classes. For example, suppose your schema contains a class named `Derived` that has a class named `Base` as its base class. You want to introduce a new class named `NewBase` whose base class is `Base`, and you want to change `Derived` to have `NewBase` as its direct base class. `Derived` still derives from `Base`, but now it does so indirectly through `NewBase`. The following figure illustrates this:



When you insert `NewBase` you want to preserve the values of all data members in an instance of `Derived`, including data members inherited from `Base`. If you simply change the class definition of `Derived` to derive from `NewBase` instead of `Base`, schema evolution does not preserve the values of the data members of `Base` in an instance of `Derived`. This is because schema evolution does not recognize that you want the effect of the following inheritances to be the same:

- `Base` inherited indirectly through `NewBase`
- `Base` inherited directly.

However, you cannot use the usual technique of adding the new base class in one schema evolution, copying the data members into the new base class, and then deleting the old base class in a second schema evolution. C++ does not allow `Derived` to have `Base` and `NewBase` as direct base classes simultaneously when `NewBase` derives from `Base`.

To insert `NewBase` while preserving the values of data members inherited from `Base` in instances of `Derived`, perform the following steps:

1 Define a temporary class to hold the data members of `Base`. For example, define the `SaveBase` class. The `SaveBase` class

- Has the same data members as `Base` and its base classes, if any
- Does not have any methods

Temporarily, make all the data members public in `Base`, its base classes if any, and `SaveBase`.

2 Generate a new application schema that

- Contains `SaveBase`
- Changes `Derived` to inherit from both `Base` and `SaveBase`.

3 Write and run a schema evolution program that evolves the database to this new application schema. The program must include a post-evolution transform function attached to `Derived`. This transform function must copy each data member of `Base`, and its base classes if any, to the corresponding data member of `SaveBase`.

   For a transform function to be attached to `Derived`, call `os_schema_evolution::augment_post_evol_transformers` with an `os_transformer_binding` that maps the class name `Derived` to the transformer function.

4 Modify the application schema by
   - Adding `NewBase`
   - Making `Derived` inherit from both `NewBase` (in place of `Base`) and `SaveBase`

5 Run the `ossg` utility to generate the new application schema.

6 Write and run a schema evolution program that evolves the database to this new application schema. The program must include a post-evolution transform function attached to `Derived`. This transform function must copy each data member of `SaveBase` to the corresponding data member of `Base`, or its base classes, if any.

   The schema evolution program must also include an untranslatable pointer handler. This handler must redirect pointers that point to the old occurrence of `Base`, its base classes, if any, or their data members to point instead to the corresponding new occurrence.

7 Modify the application schema by
   - Removing `SaveBase`
   - Changing `Derived` to have just `NewBase` as its base class
   - Changing the protection of the data members of `Base`, and its base classes if any, from public back to private or protected as desired.

8 Run the `ossg` utility to generate the final application schema.

9 Run the `ossevol` utility to evolve the database to the final application schema.

10 Modify your application to use this final application schema.

11 Build and test your application.

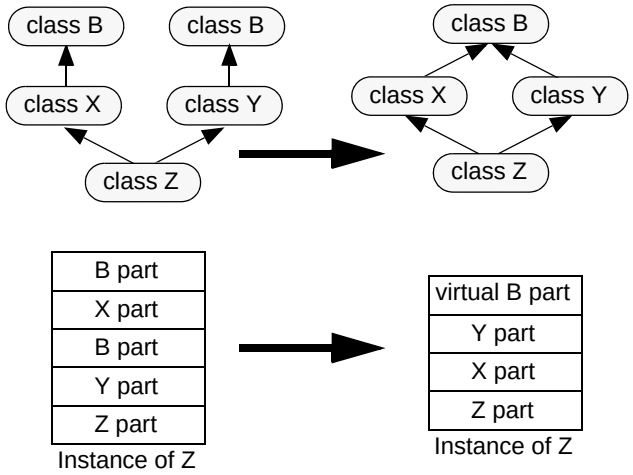
For an example of this procedure, see Example 6 in the `examples/sevol` directory of your ObjectStore installation directory.

Changing Between Virtual and Nonvirtual Inheritance

Consider a class X that inherits nonvirtually from a class B. If you change X to inherit virtually from B, instances of X must be migrated. In particular, for each instance of X, the nonvirtual B subobject is eliminated and a virtual (shared) B subobject is introduced. Each instance of X has its virtual B subobject initialized as if by a constructor that sets each member to `0`. This applies to all instances of X, including instances that are subobjects of other objects either as a data member value or as a subobject corresponding to a base class. The figure shows one such case. In general,

every virtual subobject introduced by the inheritance change is initialized as if by a constructor that sets each field to `0`.

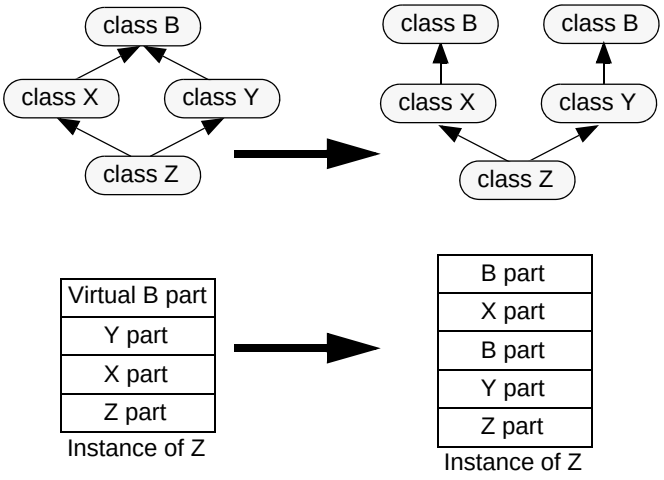
Virtual inheritance



When you change both X and Y to inherit virtually from B, instances of Z (derived from both X and Y) are migrated so that they have only a single B part.

Similarly, if inheritance is changed from virtual to nonvirtual, every nonvirtual subobject introduced by the change is initialized as if by a constructor that sets each field to 0. Therefore, if X has a virtual base class, B, changing X to inherit nonvirtually from B eliminates a virtual B subobject from each instance of X and introduces a nonvirtual B subobject that is initialized as if by a constructor that sets each member to `0`.

Nonvirtual inheritance



When you change either X or Y to inherit nonvirtually from B, instances of Z (derived from both X and Y) are migrated so that they have two B parts.

Class Deletion

Deletion of a class by using the schema evolution facility results in deletion of all the class's instances during the instance initialization phase. Pointers and references to objects deleted in this way signal `err_se_deleted_object`.

Evolving Schemas That Contain Union Bystanders

You can use the `ossevol` utility or the `os_schema_evolution` class to evolve schemas that contain unions only when those unions are not directly involved in the schema evolution. Such unions are referred to as union bystanders. To understand what a union bystander is, some background information is required.

- In the context of schema evolution of bystander unions, the term *class* includes `struct` and `union`.
- ObjectStore evolves a class if at least one of the following is true:
  - The class's definition is directly changed in the new schema.
  - Any nonstatic data member of the class has a type that is a class that is being evolved.
  - Any base class of the class is being evolved.
- A top-level persistent object
  - Is allocated with persistent `new`.
  - Is not a member or base of another object.
  - Can be an array element of a top-level array.
  - Is evolved if its class is evolved.
- A class contains a union if one of the following is true:
  - The class is the union.
  - The class has the union as a direct or indirect base class.
  - The class has a direct or indirect nonstatic data member whose type is a class that contains the union.
- A top-level persistent object contains a union if its class contains a union.
- A union contains a pointer if one of the following is true:
  - Any variant of the union is a pointer-type value.
  - Any variant of the union is a class that has a direct or indirect nonstatic data member that is a pointer-type value.

A union bystander is a union that exists in the database schema and for which at least one of the following is true:

- The database contains no persistent instances of any class that contains the union.
- You are using the `ossevol -classes_to_be_removed` argument to remove the union.
- All of the following are true:
  - You are not evolving a top-level persistent object that contains the union.
  - The union does not contain an ambiguous pointer to a top-level persistent object that is being evolved. A pointer is ambiguous if two variants of the union contain pointers at the same offset with different types.

- No top-level persistent object that contains the union contains any member of type pointer to data member where the target of the pointer is directly or indirectly inside a class that is evolved. The C++ syntax for pointer to data member is "`class::*`".
- No top-level persistent object contains two members of type pointer to data member where one points to a member directly or indirectly inside a class that is being evolved, and the other points to a member directly or indirectly inside a class that contains the union.
- The union does not have an evolution transformer function attached to the union itself or to any class that is a member or base class inside of the union.

If a union is directly involved in the schema evolution, in other words if the above rules are violated in any way, ObjectStore signals an `err_union_evolution` exception with an `err-0022-0156` error message that includes a specific explanation of the violation. ObjectStore reports the first violation that it detects and then terminates schema evolution.

At this point, schema evolution has progressed far enough so that it is impossible to use or repair the database. You must do one of the following:

- Remove objects from your database:
  - **a** Restore the database from a backup copy.
  - **b** Remove the offending object and all similar objects that would also prevent schema evolution.
  - **c** Back up the database.
  - **d** Rerun schema evolution.
- Change the new schema to be less different:
  - **a** Restore the database from a backup copy.
  - **b** Change the new schema to be less different from the old schema. Remove changes that directly involve unions.
  - **c** Back up the database.
  - **d** Rerun schema evolution.

Evolving or Compacting Very Large Databases

When you are evolving or compacting a very large database, ObjectStore might run out of processing space. If it does, it generates `err-0019-0007 err_log_data_segment_full` and terminates. To successfully evolve or compact your database, try the following steps, one at a time, until the error goes away.

- Stop other clients that are using the same ObjectStore server. Retry the interrupted operation.
- If you are running the `oscompact` utility, specify the `-work_db` argument. This lets ObjectStore compact each cluster in a separate transaction. Retry the interrupted operation.
- Decrease the number of threads, possibly all the way to 1, by specifying the `-threads` option to `oscompact` or `ossevol`, or the `threads` argument to the `initialize()` function of `os_compact` or `os_schema_evoluton`. Retry the interrupted operation.
- Decrease the value of `-maximum_cluster_size`. This requires the compaction or schema evolution operation to start again from a fresh copy of the database. Be sure that you retry the operation on a fresh database. That is, before retrying the operation, you must restore the database from a backup that you performed before starting compaction or schema evolution the first time.

  This step is only for 6.3.0 installations. In older releases, you cannot decrease the value of the maximum cluster size.

  If you did not specify a work database when you ran the `oscompact` utility, you do not need to restore your database. Your database will be unchanged because the transaction in which compaction was done was aborted. When compaction uses a work database, the operation is done in multiple transactions. Consequently, the database can be in an inconsistent state when compaction terminates because of insufficient processing space.

If none of these steps solves the problem, contact technical support.

Chapter 8
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.

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 dest, 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 ! */
};
```

Example

Following 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:

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!
  }
```

Caution

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 because, typically, incoming text is all in the single encoding defined by the operating system used when generating it. Autodetect can be used only at the beginning of a session and it is 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. Technical Support believes that the SJIS interpretation is correct for most cases.

Japanese developers are aware of the problems in 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.

- 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.
- 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 it 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, following 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 2-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

Following 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, 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* src);`

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 you are converting characters because of the additional lookup.

Restrictions

The following restrictions apply.

- 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 like a shortcoming, 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.

Index

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

X

X