# OBJECTSTORE

## ADVANCED C++ API
## USER GUIDE

RELEASE 5.1

**March 1998**

***ObjectStore Advanced C++ API User Guide***

ObjectStore Release 5.1 for all platforms, March 1998



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

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

# Contents

# Preface

Purpose

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

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

Audience

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

Scope

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

## How This Book Is Organized

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

Most of the chapters of this book parallel the chapters in the *ObjectStore C++ API User Guide*, providing a more advanced look at the ideas and features of ObjectStore, such as persistence, transactions, threads, and collections. The remainder of the chapters describe sophisticated features not generally used in more basic applications; these chapters describe queries and indexes, compaction, metaobject protocol, and the database utility

API. This publication also organizes the ObjectStore API into groups of related functions and macros.

## Notation Conventions

This document uses the following conventions:

| *Convention* | *Meaning* |
|---|---|
| **Bold** | Bold typeface indicates user input or code. |
| Sans serif | Sans serif typeface indicates system output. |
| *Italic sans serif* | Italic sans serif typeface indicates a variable for which you must supply a value. This most often appears in a syntax line or table. |
| *Italic serif* | In text, italic serif typeface indicates the first use of an important term. |
| [ ] | Brackets enclose optional arguments. |
| { *a* \| *b* \| *c* } | Braces enclose two or more items. You can specify only one of the enclosed items. Vertical bars represent OR separators. For example, you can specify *a* or *b* or *c*. |
| ... | Three consecutive periods indicate that you can repeat the immediately previous item. In examples, they also indicate omissions. |
|   | Indicates that the operating system named inside the circle supports or does not support the feature being discussed. |

## ObjectStore Release 5.1 Documentation

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

Your use of ObjectStore documentation depends on your role and level of experience with ObjectStore. You can find an overview description of each book in the ObjectStore documentation set at

URL **http://www.objectdesign.com**. Select **Products** and then select **Product Documentation** to view these descriptions.

## Internet Sources of More Information

World Wide Web

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

Internet gateway

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

Automatic email notification

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

## Training

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

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

## Your Comments

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

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

You can also fax your comments to 781.674.5440.

# Chapter 1
# Advanced Persistence

The information in this chapter augments Chapter 2, Persistence, in the *ObjectStore C++ API User Guide*. The material is organized in the following manner:

# ObjectStore Pvars

When a pointer to persistent memory is assigned to a transiently allocated variable, the value of the variable is valid only until the end of the transaction in which the assignment was made. Using database entry points typically involves looking up a root and retrieving its value — a pointer to the entry point. Frequently this pointer is assigned to a transiently allocated variable for future use. However, its use is limited, since it normally will not remain valid in subsequent transactions (but see Retaining Pointer Validity Across Transactions on page 26).

Example of loss of pointer validity outside transaction

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

void f() {

    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();
}
```

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"

main() {

    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();
}
```

## Using Pvars to Maintain Pointer Validity

A convenient alternative is to use ObjectStore *pvars*. ObjectStore pvars allow you to maintain, across transactions, a valid pointer to an entry point object.

To use pvars, you define the variable you want to hold the pointer to the entry point. Then you pass the variable's address to the function **os_pvar::os_pvar()**, along with the name of the root that points to the desired entry point object, and a pointer to the database containing the root.

This function is the constructor for the class **os_pvar**, but you never have to explicitly use the instance of **os_pvar** that results. This instance must be a stack object, however, so do not create it with **new**.

Once you have called the **os_pvar** constructor, ObjectStore automatically maintains an association between the variable and the entry point. At the beginning of each transaction in the current process, if the database containing the specified root is open, ObjectStore establishes a valid pointer to the entry point object as the value of the variable. It also sets the variable to point to the entry point when the database becomes open during a transaction.

When control leaves the block in which the **os_pvar** constructor was called, the destructor for the resulting instance of **os_pvar** is executed, breaking the association between the variable and entry point. Therefore, if you are using the ObjectStore transaction macros, you should call the constructor from outside any transaction, since the macros establish their own block.

## Additional Type Safety

As with **os_database_root::get_value()**, you can also supply an **os_typespec*** to **os_pvar::os_pvar()** for additional type safety. ObjectStore will check that the specified typespec matches the typespec stored with the root. Note that it checks only that the typespec supplied matches the stored typespec, and does not check the type of the entry point object itself.

See Type Safety for Database Roots in Chapter 2 of the *ObjectStore C++ API User Guide* for more information on the use of typespecs.

## Pvar Example

Here is an example of the use of pvars:

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

void f() {

    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_pvar p(db1, &a_part_p, "part_root", &part_type);

    OS_BEGIN_TXN(tx1,0,os_transaction::update)
        . . .
    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();
}
```

Note that, even though you can use this variable from one transaction to the next without re-retrieving its value, you cannot use it *in between* transactions. As always, you must be within a

transaction to access persistent data. In between transactions, ObjectStore automatically sets the variable to **0**. The variable is also set to **0** during a transaction if the database containing its associated root is closed.

Note also that you should not try to set the value of this variable, since ObjectStore handles all assignment of values to it. If you want to retrieve different objects at different times through the use of a single pvar, use a pvar to associate a pointer-valued variable with a pointer-valued entry point object. Then you can change what the entry point points to as needed.

## Initialization Functions

You can also create an entry point and root using **os_pvar::os_pvar()**, by supplying a pointer to an initialization function. The function should allocate the entry point object in a given database, and return a pointer to the new object.

This function will be executed upon the call to **os_pvar()** or at the beginning of subsequent transactions, if the database to contain the root is open and ObjectStore cannot find the specified root in that database. It will also be called when this database becomes open during a transaction, and ObjectStore cannot find the root in that database.

### **os_pvar** constructor declaration

Here is how the **os_pvar** constructor is declared:

```
os_pvar(
    os_database *db,
    void *location,
    char *root_name,
    os_typespec *typespec = 0,
    void *(*init_fn)(os_database*) = 0
);
```

ObjectStore provides three standard initialization functions, **os_pvar::init_long()**, **os_pvar::init_int()**, and **os_pvar::init_pointer()**. These each allocate an object of the appropriate type (**long**, **int**, or **void***), initialize the object to **0**, and return a pointer to it. You can supply either a standard or a user-defined function to the **os_pvar** constructor.

### Example pvar initialization function

Here is an example of the use of a pvar initialization function:

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

```
void *part_init(database *db) {
    static os_typespec part_type("part");
    return new(db, part_type) part("part_0");
}

void f() {

    objectstore::initialize();

    static os_typespec part_type("part");
    part *a_part_p = 0;
    employee *an_emp_p = 0;
    database *db1 = database::open("/thx/parts");
    os_pvar p(db1, &a_part_p, "part_root", &part_type, part_init);

    OS_BEGIN_TXN(tx1,0,os_transaction::update)
        a_part_p->display();
        . . .
    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();
}
```

# Creating Object Clusters

Object clusters, like segments, are created explicitly. Just as you create a segment by performing **os_database::create_segment()** on the database to contain the new segment, you create an object cluster by performing **os_segment::create_object_cluster()** on the segment to contain the new cluster.

**os_object_cluster* create_object_cluster(os_unsigned_int32 size) ;**

The function returns a pointer to an **os_object_cluster**. Unlike segments, however, which are variable sized and expand to accommodate whatever is added to them, clusters have a fixed size. You specify the size in bytes of a new cluster as an argument to **create_object_cluster()**. This number must be less than 65536, since 64 KB is the maximum cluster size.

```
os_segment *seg1;
    . . .
os_object_cluster *clust1 = seg1–>create_object_cluster(4096) ;
```

The actual size of the new cluster is the result of rounding the specified size up to the next whole number of pages, minus the platform architecture alignment *(*see *ObjectStore Building C++ Interface Applications*).

Do not perform **create_object_cluster()** on the transient segment.

## Allocating a new object within an existing cluster

You can use **os_object_cluster::of()** to allocate a new object in the same cluster as an existing one:

```
os_database *db1;
part *an_old_part,

. . .

part *a_new_part = new(
        os_object_cluster::of(an_old_part) ,
        part_type
) part(111);
```

# Setting Data Fetch Policies

An ObjectStore application can control, for each segment, 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 *just* 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 between fetch policies

ObjectStore has several *fetch policies* you can associate with a given segment to control transfer granularity:

- **os_fetch_segment** specifies that the entire segment 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 segment. It is useful for special applications scanning large quantities of data.

## Specifying a fetch policy

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

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

void os_database::set_fetch_policy (
    os_fetch_policy, os_int32 bytes);

void os_segment::set_fetch_policy (
    os_fetch_policy, os_int32 long bytes);
```

Using the **set_fetch_policy()** function on an **os_database** object changes the fetch policy for all segments in that database, including segments created by the current process in the future.

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

concurrent processes can use a different fetch policy for the same segment or database.

## os_fetch_segment Policy

For applications that manipulate substantial portions of small segments, the **os_fetch_segment** policy is appropriate. Under this policy, ObjectStore attempts to fetch the entire segment containing the desired page, in a single client/server interaction, if the segment will fit in the client cache without evicting any other data. If there is not enough space in the cache to hold the entire segment, the specified number of bytes are fetched, rounded up to the nearest positive number of pages. (Note that if you specify **0** bytes, this will be rounded up, and the unit of transfer will be a single page.) The **os_fetch_segment** policy is very efficient if a significant portion of the segment will be required, but wastes time and bandwidth if only a few pages will be referenced.

## os_fetch_page Policy

If your database contains segments larger than the client cache of your workstation, or if your application does not refer to a significant portion of each segment 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_segment *text_segment;

/* The text segment 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_segment->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**, this fetch policy lets you specify the amount of data to fetch in each client/server

interaction for a particular segment. But, in addition, it specifies that a double buffering policy should be used to stream data from the segment.

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

```
os_segment *image_segment;

/* The image segment 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_segment->set_fetch_policy (os_fetch_stream, 32768);
```

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

## 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 ObjectStore References

ObjectStore *references* provide an alternative to using pointers. ObjectStore references allow you to override default restrictions on both referring across databases and referring across transactions. References serve as substitutes for pointers, and you can usually use them as if they actually *were* valid pointers.

ObjectStore references carry some extra cost over the use of pointers. They are larger than pointers (between 8 and 16 bytes, depending on what kind you use), and dereferencing one usually involves a table lookup.

The most generally useful ObjectStore references are instances of the parameterized class **os_Reference**. More specialized reference classes are discussed in


- Using Nonparameterized References on page 17
- References and Relative Pathnames on page 19
- ObjectStore Lightweight References on page 20
- ObjectStore Protected References on page 23


Example: **os_Reference**

Here is an example of using an **os_Reference**:

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

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

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

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

    static os_database *db1 = os_database::open("/thx/parts");
    static os_database *db2 = 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(db2, 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();
}
```

Here, the member **responsible_engineer** is declared to be of type **os_Reference<employee>**. The class name in angle brackets is the *referent type*. It indicates that values of **responsible_engineer** are references to instances of the class **employee**.

When the **employee*** (**an_emp**) is assigned to **a_part->responsible_engineer**, a reference to this **employee*** is automatically constructed and stored there. You can use an **employee*** anywhere a **reference<employee>** is expected. In general, you can use a **T*** anywhere a **reference<T>** is expected, because there is a conversion constructor, **os_Reference::os_Reference(T*)**.

Now you can use the reference to the employee in many contexts requiring an **employee***. This is because the class **os_Reference** overloads the **–>** operator, and defines a conversion operator so its instances are converted to pointers to instances of its referent type, when appropriate. So you just use the reference as you would a pointer to its referent type, as in the **printf** statement.

Note that the **->** and conversion operators are the only operators with special reference class overloadings, so references do not behave like pointers in the context of other operators, like **[]** and **++**.

## Automatic Database Open

If an ObjectStore reference refers to an object in a database that is not open, ObjectStore opens the database automatically when the object is accessed.

## Using memcpy() with Persistent os_references and Related Classes

You can use the C++ **memcpy()** function to copy a persistent **os_reference** only if the target object is in the same segment as the source object. This is because all persistent **os_reference**s use **os_segment::of(this)** for **os_reference** resolution processing and the resolution will be incorrect if the **os_reference** has been copied to a different segment. This restriction holds true for the eponymous types of the parameterized and unparameterized versions of the

following classes: **os_Reference**, **os_Reference_protected**, and **os_Reference_ this_DB.**

## Resolution by Relative Pathname

As with cross-database pointers, instances of **os_Reference** store a relative pathname to identify the referent database. See References and Relative Pathnames on page 19.

## Referring Across Transactions

"Example: os_Reference" on page 11 shows how references can be used to refer from one database to another. References can also be used to refer across transactions.

In an ObjectStore application, you typically retrieve pointers to persistent objects and store them in transiently allocated variables. You can normally use these pointers only in the transaction in which they were retrieved:

Example of transiently allocated (invalid) pointers

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

void f() {

    objectstore::initialize();

    os_Set<part*> *part_set;
    part *a_part;

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

    OS_BEGIN_TXN(tx1, 0, os_transaction::update)
        part_set = (os_Set<part*>*) (
            db1->find_root("part_set")–>get_value()
        ); /* retrieval */
        . . .
    OS_END_TXN(tx1)

    OS_BEGIN_TXN(tx2, 0, os_transaction::update)
        a_part = part_set->query_pick(   /* INVALID! */
            "part", "part_number==123456", db1
        );
        . . .
    OS_END_TXN(tx2)

    db1–>close();
}
```

Here, **part_set** is a pointer to an entry point, a set of **part*** pointers. Since **part_set** is transiently allocated, its value is valid only until

the end of the current transaction. So its use in the query in the next transaction is invalid, and will have unpredictable results. If you want to use this pointer in a subsequent transaction, it must be retrieved again:

Example of retrieving a previously allocated pointer

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

main() {

    objectstore::initialize();

    os_Set<part*> *part_set;
    part *a_part;

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

    OS_BEGIN_TXN(tx1, 0, os_transaction::update)
        part_set = (os_Set<part*>*) (
            db1->find_root("part_set")–>get_value()
        ); /* retrieval */
        . . .
    OS_END_TXN(tx1)

    OS_BEGIN_TXN(tx2, 0, os_transaction::update)
        part_set = (os_Set<part*>*) (
            db1->find_root("part_set")–>get_value()
        ); /* re-retrieval */

        a_part = part_set->query_pick(   /* OK */
            "part", "part_number==123456", db1
        );
        . . .
    OS_END_TXN(tx2)

    db1–>close();
}
```

Using references to avoid re-retrieving previously allocated pointers

But suppose retrieving the required pointers is relatively complicated or expensive, and you need to use them in many transactions. Then it might be preferable to use ObjectStore references:

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

main() {

    objectstore::initialize();
    part *a_part;
    os_Reference<os_Set<part*>    > part_set_ref;
    os_database *db1 = os_database::open("/thx/parts");
```

```
    OS_BEGIN_TXN(tx1, 0, os_transaction::update)
        part_set_ref = (os_Set<part*>*) (
            db1->find_root("part_set")–>get_value()
        ); /* retrieval */
        . . .
    OS_END_TXN(tx1)

    OS_BEGIN_TXN(tx2, 0, os_transaction::update)
        a_part = part_set_ref->query_pick(
            "part", "part_number==123456", db1
        ); /* OK */
        . . .
    OS_END_TXN(tx2)

    db1–>close();
}
```

Here, the variable **part_set_ref** is declared to be of type **os_Reference< os_Set<part*> >**. The class name in angle brackets, **os_Set<part*>**, is the referent type. It indicates that **part_set_ref** refers to an instance of the class **os_Set<part*>**. When the **os_Set<part*>*** returned by the **get_value()** is assigned to **part_set_ref**, a reference to the **os_Set<part*>** is automatically constructed and stored in **part_set_ref**. You can use an **os_Set<part*>*** anywhere an **os_Reference< os_Set<part*> >** is expected.

As described above, you can use a **T*** anywhere a **os_Reference<T>** is expected, because there is a conversion constructor, **os_Reference::os_Reference(T*)**.

You can also use the **os_Reference** in any context requiring an **os_Set<part*>**. As mentioned above, this is because the class **os_Reference** overloads the ***** and **–>** operators, and defines a conversion operator so its instances are converted to pointers to instances of its referent type, when appropriate. So you just use the reference as you would a pointer to its referent type, as when calling **query_pick()**.

# Generating One Reference from Another

A reference can be used to generate a text stream, which in turn can be used to generate another reference to the same object. You do this using **::operator <<()** and **::operator >>()**. For example:

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

void dump_part(part *a_part) {
    os_Reference<part> part_ref = a_part;
    cout << part_ref;
}

os_Reference<part> read_dump() {
    os_Reference<part> part_ref;
    cin >> part_ref;
    return part_ref;
}
```

# Using Nonparameterized References

If your compiler does not support class templates, you can use the nonparameterized reference class **os_reference**. You also should use **os_reference** if you need a reference to an instance of a built-in type like **int** or **char**, since the referent type of an **os_Reference** must be a class.

**os_reference** is just like **os_Reference**, except the conversion constructor used is **os_reference(void*)** instead of **os_Reference(T*)**. In addition, the conversion operator used is **operator void*()** instead of **operator T*()**, which means that you should use a cast to pointer-to-referent type when dereferencing an **os_reference**.

### Example: using nonparameterized references

Here are some examples:

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

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

class part {
    static os_typespec *get_os_typespec();
    . . .
    os_reference responsible_engineer;
    . . .
};

f() {
    objectstore::initialize();

    static os_database *db1 = os_database::open("/thx/parts");
    static os_database *db2 = 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(db2, employee::get_os_typespec()) employee;
        a_part->responsible_engineer = an_emp;

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

    OS_END_TXN(tx1)
    db1–>close();

}
```

Example: using nonparameterized references

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

main() {

    objectstore::initialize();

    part *a_part;

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

    os_reference part_set_ref;

    OS_BEGIN_TXN(tx1, 0, os_transaction::update)
        part_set_ref = (os_set*) (
            db1->find_root("part_set")–>get_value()
        ); /* retrieval */
        . . .
    OS_END_TXN(tx1)

    OS_BEGIN_TXN(tx2, 0, os_transaction::update)
        a_part = (part*) (
            ((os_set*) (part_set_ref))->query_pick(
                "part", "part_number==123456", db1
            )
        ) ; /* ok */

        . . .

    OS_END_TXN(tx2)
    db1–>close();

}
```

# References and Relative Pathnames

As is true with cross-database pointers, instances of **os_Reference** store a relative pathname to identify the referent database. The pathname is relative to the lowest common directory in the pathnames of the referent database and the database containing the reference. For example, if a reference is stored in **/A/B/C/db1** that refers to data in **/A/B/D/db2**, the lowest common directory is **A/B**, so the relative pathname **../../D/db2** is used.

This means that if you copy a database containing an **os_Reference** or **os_reference**, the reference in the copy and the reference in the original might refer to different databases. To change the database a reference refers to, you can use the ObjectStore utility **oschangedbref**, which is described in oschangedbref: Changing External Database References in *ObjectStore Management*.

Note that an intradatabase reference (a reference referring to an object in the same database) uses a pathname of the form **../db-name**, where **db-name** is the terminal component of the database's pathname. So if you copy or move this database in such a way as to change the terminal component of its pathname — say by making a copy within the same directory — the reference in the copied or moved database will not be an intradatabase reference; it will refer to the location **../db-name**, which has the same terminal component as the original. To make sure you get an intradatabase reference, use **os_Reference_this_DB** or **os_Reference_local.** See ObjectStore Lightweight References on page 20.

A discussion on how relative pathnames work can be found in Cross-Database Pointers and Relative Pathnames in Chapter 2 of the *ObjectStore C++ API User Guide.*

# ObjectStore Lightweight References

Instances of the class **os_Reference** take up 12 bytes of storage each. If you are using a large number of references, you might be able to use one of the lightweight reference types to save space.

There are three kinds of lightweight references:

- Instances of **os_Reference_local** or **os_reference_local** (8 bytes)
- Instances of **os_Reference_transient** or **os_reference_transient** (8 bytes)
- Instances of **os_Reference_this_DB** or **os_reference_this_DB** (8 bytes)

A summary of the various ObjectStore reference classes is presented in the tables in Summary of ObjectStore Reference Types on page 24.

## Local References

Each **os_Reference** stores a database pathname. An **os_Reference_local**, in contrast, saves space by not storing the pathname of the referent database, and instead requiring you to supply a pathname when dereferencing it. So you dereference an **os_Reference_local** explicitly, with a call to the function **os_Reference_local::resolve()**, which takes a **database*** argument.

### Example: use of **os_Reference_local**

Here is an example:

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

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

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

f() {
    objectstore::initialize();

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

```
    static os_database *db2 =
        os_database::open("/thx/employees");

    OS_BEGIN_TXN(tx1, 0, os_transaction::update)
        part *a_part = new(db1, part::get_os_typespec()) part;
        employee *an_emp =
            new(db2, employee::get_os_typespec()) employee;
        a_part->responsible_engineer = an_emp;

        printf("%d\n",
          a_part->responsible_engineer->resolve(db2)->emp_id);
    OS_END_TXN(tx1)

    db1–>close();
}
```

Just as **os_reference** is the nonparameterized version of **os_Reference**, **os_reference_local** is the nonparameterized version of **os_Reference_local**. Resolving an **os_reference_local** is just like resolving an **os_Reference_local**; no cast is necessary (as is necessary with an **os_reference**), since resolution is explicit.

## Using Transient References with os_Reference_transient

Instances of **os_Reference_transient** are used to refer across transactions, but can only refer from transient memory. So if you need a reference that can be transiently allocated, you can use an **os_Reference_transient**. No explicit resolution is necessary, since such a reference stores a **database*** (which saves space compared to storing a database pathname). You use it just as you would an **os_Reference**, except that it cannot be allocated in persistent storage (since the **database*** it contains actually points to a transient object).

There is also a nonparameterized class, **os_reference_transient**, for referring from transient memory. You use it just as you would **os_reference**, except that it cannot be allocated in persistent storage.

## Reducing Relocation Overhead

Using ObjectStore references has an advantage not yet discussed: they can reduce the amount of virtual memory that ObjectStore reserves during certain transactions. When ObjectStore retrieves the objects in a particular database segment, it reserves virtual memory addresses for all the objects in segments *pointed to* by objects in the retrieved segment. In other words, virtual memory is preassigned corresponding to each outgoing pointer in the retrieved segment (where an outgoing pointer points to an object

in a different segment, not necessarily a different database). Whenever an outgoing ObjectStore reference is used in a segment instead of an outgoing pointer, this reduces the amount of virtual memory that must be reserved by transactions that retrieve the segment.

If you want to use a reference solely for the purpose of saving on virtual memory addresses, you can use an **os_Reference_this_DB**. When ObjectStore resolves one of these references, it assumes the referent is in the same database as the reference itself. So, even though these references save space by not storing a database pathname, they do not have to be resolved explicitly. You use an **os_Reference_this_DB** just as you would an **os_Reference**.

There is also a nonparameterized class **os_reference_this_DB**, which you use just as you would **os_reference**.

# ObjectStore Protected References

You can ensure referential integrity for ObjectStore references by using *protected* versions of the reference classes. Once an object referred to by a protected reference is deleted, use of the protected reference causes an err_reference_not_found exception to be signaled. If the referent database has been deleted, err_database_not_found is signaled. If you do not use protected references, then, as with regular C++ pointers, you can access arbitrary memory by using a reference whose referent has been deleted.

The protected reference classes are

- **os_Reference_protected** (used just as is **os_Reference**)
- **os_Reference_protected_local** (used just as is **os_Reference_local**)

Do not use protected references to refer to transient memory.

Each time you create a protected reference, a write to the database is performed, to maintain a persistent table that associates protected references with their referents.

You can test a safe reference to see if its referent has been deleted with the member function **deleted()**.

A summary of the various ObjectStore reference classes is presented in Summary of ObjectStore Reference Types on page 24.

# Summary of ObjectStore Reference Types

Parameterized reference classes

The table below summarizes the characteristics of the various parameterized ObjectStore references.

| *Class* | *Purpose* | *Resolution* | *Reference Allocation* | *Referent Allocation* | *Size in Bytes* |
|---|---|---|---|---|---|
| **os_Reference** | Refer across databases and transactions | Implicit | Anywhere | Anywhere | 12 |
| **os_Reference_local** | Refer across databases and transactions | **resolve( referent-database*)** | Anywhere | Anywhere | 8 |
| **os_Reference_transient** | Refer across transactions | Implicit | Transient memory | Anywhere | 8 |
| **os_Reference_this_DB** | Save on VM addresses | Implicit | Same database as referent | Same database as reference | 8 |
| **os_Reference_protected** | Refer across databases and transactions; referential integrity | Implicit; signals error if referent deleted | Anywhere | Persistent memory | 12 |
| **os_Reference_protected_local** | Refer across databases and transactions; referential integrity | **resolve( referent-database*)** signals error if referent deleted | Anywhere | Persistent memory | 8 |

### Nonparameterized reference classes

The table below summarizes the characteristics of the various nonparameterized ObjectStore references.

| *Class* | *Purpose* | *Resolution* | *Reference Allocation* | *Referent Allocation* | *Size in Bytes* |
|---|---|---|---|---|---|
| **os_reference** | Refer across databases and transactions | Cast to pointer to referent type | Anywhere | Anywhere | 12 |
| **os_reference_local** | Refer across databases and transactions | **resolve( referent-database*)** | Anywhere | Anywhere | 8 |
| **os_reference_transient** | Refer across transactions | Cast to pointer to referent type | Transient memory | Anywhere | 8 |
| **os_reference_this_DB** | Save on VM addresses | Cast to pointer to referent type | Same database as referent | Same database as reference | 8 |
| **os_reference_protected** | Refer across databases and transactions; referential integrity | Cast to pointer to referent type; signals error if referent deleted | Anywhere | Persistent memory | 12 |
| **os_reference_protected_local** | Refer across databases and transactions; referential integrity | **resolve( referent-database*)**; signals error if referent deleted | Anywhere | Persistent memory | 8 |

# Retaining Pointer Validity Across Transactions

If you are converting an existing application to use ObjectStore, it might be inconvenient to rewrite the code to use references, especially if the application will use many short transactions. ObjectStore has a feature that enables you to retain persistent addresses across transaction boundaries, so that it is not necessary to use references.

The advantage of this feature is that code is easier to port to ObjectStore. The disadvantage is that ObjectStore might run out of available persistent addresses if too much persistent data is referenced. This is because ObjectStore normally unmaps all persistent data from virtual memory at the end of each transaction. This feature disables that unmapping. 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.

The static member function **objectstore::release_persistent_addresses()** globally releases all persistent addresses. After this function is called, all existing transient-to-persistent pointers are invalidated.

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: retrieving persistent addresses

Here is an example:

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

main() {

    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)

}
```

# Discriminant Functions

For each union type intended to have persistent instances, you must supply an associated *discriminant function*. Discriminant functions allow ObjectStore to determine the actual layout of any persistent object when mapping it into memory.

Consider the following union:

```
union myunion {
    struct {
        int i;
        foo * fptr;
    } S1 ;

    struct {
        bar * btr;
        float f;
    } S2 ;

} ;
```

ObjectStore sometimes has the task of modifying all pointers embedded in an object when the object is brought into memory (see ObjectStore Memory Mapping Architecture in Chapter 1 of the *ObjectStore C++ API User Guide*). In the case above, such an object will contain a pointer either at offset 0 or at offset 4 bytes, depending on whether **S1** or **S2** is the correct interpretation of the object's structure.

Because the application can reconfigure the object (switch between **S1** and **S2**) at will, the application must record the state of the object relative to layout, and provide a functional interface for extracting that information. This functional interface is the discriminant function.

## Defining and using unions with discriminant functions

The union above can be modified slightly to accommodate this as follows:

```
union myunion {
    int Tag;

    struct {
        int T;
        int i;
        foo * fptr;
    } S1 ;

    struct {
```

```
        int T;
        bar * btr;
        float f;
    } S2 ;
    os_int32 discriminant();
} ;
os_int32 myunion::discriminant() { return Tag; }
```

Applications that use this union type must take care to do

```
S1.T = 1;
```

when they want to use the **S1** layout.

Likewise, when they want to use the **S2** layout, they should do

```
S2.T = 2;
```

Switching layouts must be accompanied by reassigning the leading integer.

### Example: defining and using a class with a union-valued data member

You can define a class with a union-valued data member as follows:

```
class myclass {
    public:
    myunion MyU;
    int Tag;
    os_int32 discriminant();
};
os_int32 myclass::discriminant() { return Tag; }
```

Now applications must take care that the **Tag** records the value of the union layout. For example:

```
myclass MyC;
MyC.Tag = 1; MyC.S1.i = 0;
/* . . . later on we switch */
MyC.Tag = 2; MyC.S2.f = 1.1;
```

In the first example, the name of the discriminant function, **myunion::discriminant**, serves to associate the function with the union **myunion**. In the second example, the name **myclass::discriminant** serves to associate the function with the one and only union-valued data member of **myclass**. When a class has more than one union-valued data member, the name of each discriminant function should have the form

**discriminant_union-name_data-member-name**

where **union_name** is the name of the associated union, and **data-member-name** is the name of the associated data member.

For heterogeneity considerations, in the *ObjectStore C++ API Reference*, Chapter 5 see Discriminant Functions for additional details.

# Chapter 2
# Advanced Transactions

This chapter is intended to augment Chapter 3, Transactions, of the *ObjectStore 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 will both be 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 will not be locked together.

## Locking Granularity

Another way to help avoid locking data unnecessarily is to avoid unnecessary use of segment-level locking granularity; that is, to avoid unnecessary use of **lock_segment_read** or **lock_segment_write** as the argument to **os_segment::set_lock_whole_segment()**. Unnecessary use of **lock_segment_write** can also increase the amount of data transferred out of the client cache. The benefit of segment granularity locking is that it avoids the overhead of a separate page fault for each page locked, and it can reduce Server communication.

For more information, see **os_segment::set_lock_whole_segment()** in Chapter 2 of the *ObjectStore C++ API Reference*.

## 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 will be in a consistent state between transactions (see Nested Transactions on page 34).

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.

It is sometimes particularly important to make transactions that use persistent **new** or persistent **delete** as short as possible.

## Multiversion Concurrency Control (MVCC)

Single-database, read-only transactions can use *multiversion concurrency control*, or MVCC. When you use 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. See Multiversion Concurrency Control (MVCC) on page 37 for additional information.

## abort_only Locking Rules

The locking restrictions are relaxed somewhat when the transaction is **abort_only**. Under such circumstances, the client does not get write locks for any pages that are written during an abort-only 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. This lock relaxation is another method of reducing wait time.

## Lock Timeouts

*Lock timeouts* provide the ability to limit wait time, and 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. The **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 *ObjectStore C++ API Reference*.

# Nested Transactions

### Why use 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 both from inside and outside a transaction.

### Nested transactions must be of the same type

Except when you are using **os_transaction::abort_only**, when you nest one transaction within another, the two transactions must be of the same type (**os_transaction::update** or **os_transaction::read_only**). If they have different types, err_trans_wrong_type is signaled.

### Nested transactions and **abort_only**

When you are using **os_transaction::abort_only**, if the top-level transaction is **abort_only**, then both **abort_only** and **update** transactions can nest within it.

Note that an **abort_only** transaction does not automatically abort. You must specifically use the **os_transaction::abort()** function to abort the **abort_only** transaction, otherwise an exception is signaled.

You can use **abort_top_level()**, or for stack transactions use **abort()**, since you know exactly where the transaction ends.

For example:

```
OS_BEGIN_TXN(txn, 0, os_transaction::abort_only) {
    . . .
    os_transaction::abort();
} OS_END_TXN(txn);
```

When a nested transaction is aborted, persistent data is rolled back to its state as of the beginning of that transaction. However, no locks are released until the outermost transaction terminates. This means other processes still have to wait to access the pages that the aborted transaction accessed.

# Deadlock

ObjectStore sometimes automatically aborts a transaction due to 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 46. There are other, more complicated forms of deadlock that are analogous.

## Deadlock Victim

ObjectStore has a deadlock detection facility that breaks deadlocks, when 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 other processes can proceed.

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

## Automatic Retries Within Lexical Transactions

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

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

Its default value is **10**. You can change the value of **os_transaction::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. But 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 3 of the *ObjectStore 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 multiversion concurrency control.

If a transaction

- Only performs read access on a database, and
- 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 should open the database for multiversion concurrency control (MVCC). You can do this with members of the class **os_database** (see The MVCC API on page 38). You can also use MVCC in conjunction with **os_transaction::abort_only**. This can improve your application's performance, as well as the performance of other concurrent ObjectStore applications.

## 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 47. 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 47.

## 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.

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.

## Accessing Multiple Databases in a Transaction

When an application reads a database opened for MVCC, the snapshot it sees is potentially out of date. This means that the snapshot might not be consistent with other databases accessed in the same transaction (although it will always be internally consistent). Even two databases both of which are opened for MVCC might not be consistent with each other, because updates might be performed on one of the databases in between the times of their snapshots.

## Serializability

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

## The MVCC API

You open a database for multiversion concurrency control with one of the following members of **os_database**:

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

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

Once you open a database for MVCC, multiversion concurrency control 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 opened for MVCC with the following member of **os_database**:

**os_boolean is_open_mvcc() const ;**

This function returns nonzero if this is opened for MVCC, and **0** otherwise.

## MVCC and the Transaction Log

Although multiversion concurrency control can cause ObjectStore to, in effect, take a snapshot of an entire database, the implementation actually only copies data 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 41.

In the absence of multiversion concurrency control, 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

Note that this means that long transactions that use multiversion concurrency control can cause the log to become very large.

### Conflict detection

Multiversion concurrency control determines whether a page must be held in the log based on the notion of *conflict* defined below. 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 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 48.

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.

# 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 en route 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 below), 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 someone pulls the plug 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 in order 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 *ObjectStore Management*.

# Checkpoint: Committing and Continuing a Transaction

ObjectStore includes a way to perform a checkpoint within a transaction. The checkpoint commits modified data from a top-level transaction without incurring the overhead of ending a transaction and starting a new transaction. This done with the **os_transaction::checkpoint()** interface.

The **os_transaction::checkpoint()** interface is similar to **os_transaction::commit()**, with the difference that you get the effect of committing a transaction and then continuing work in a new transaction in which you have read locks on all or most of the persistent objects that were locked in the committed transaction. This is useful when

- You are making modifications to a database. You want to periodically commit your changes but continue updating the database without intervention. For example, you might be loading new data into the database.
- You want to make your changes available to MVCC readers.
- You opened a database for MVCC and you want an updated snapshot.

Note

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. The details are explained in the next section.

Caution

Checkpoint allows the application to maintain lock assertion, relocation state, address space assignment, and page protection state for some pages in the cache across what ordinarily would be transaction boundaries. This means that for certain classes of applications that access the same data pages repeatedly in sequential transactions, you can avoid the cost of setting up and tearing down access to those pages repeatedly. Like transaction commit and abort operations, this operation is not thread-safe. Applications must ensure that other threads do not access persistent memory during a checkpoint operation.

In conjunction with MVCC-opened databases, checkpoint can also be used to expose to the current transaction changes that have

been committed to the databases since the transaction started (or since the last checkpoint invocation). This brings the transaction up to date with changes that have taken place without its knowledge.

See **os_transaction::checkpoint()** and **os_transaction::checkpoint_in_progress()** in Chapter 2, Class Library, of the ObjectStore C++ API Reference for further detail.

## Advantages of a Checkpoint

The advantage of a checkpoint is that there is less overhead than when you actually end one transaction and start another. When you checkpoint a transaction, it is as if you committed the transaction and then immediately started a new transaction. But in the new transaction, you already have read locks on most or all of your persistent objects.

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.

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.

After a checkpoint, ObjectStore has read locks on the same objects as before the checkpoint, unless another client was waiting for a write lock on one of these objects. In that case, your transaction loses the lock.

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.

## Calling the os_transaction::checkpoint() Function

To checkpoint a transaction, call the **os_transaction::checkpoint()** function. The function's overloadings are

- **static void os_transaction::checkpoint();**

  Invokes checkpoint on the current transaction.

- **static os_transaction::checkpoint(os_transaction*);**

  Invokes checkpoint on the specified transaction.

Caution

Before you checkpoint a transaction, you must ensure that the database is in a consistent state.

During the checkpoint, you must ensure that no other thread tries to access the database.

For related information, see **os_transaction::checkpoint_in_progress()**.

# 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, it is not blocked. But if the latter 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 this:

| *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**. But Transaction 1 cannot proceed until Transaction 2 completes and releases its lock on **P2**. Since 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 46.

| ***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 multiversion concurrency control, deadlock would have resulted. See the Simple Deadlock Scenario on page 46**.**

| ***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 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
# Threads

ObjectStore supports the use of multiple threads within a client application. The key to developing a successful multithreaded application with ObjectStore is in choosing the right combination of transaction, process, and thread models. This is evident once you have good understanding of the process requirements and ObjectStore's implementation of transactions and how it accomplishes thread safety.

# ObjectStore Thread Safety

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

While ObjectStore supports multithreaded clients, it currently supports only one independent transaction per process. However, multithreaded applications can choose any of the following models for interacting with ObjectStore:

- One thread dedicated to ObjectStore access
- Multiple threads accessing ObjectStore and sharing a transaction
- Multiple threads accessing ObjectStore in separate transactions but also doing other activities (using local transactions, but limiting the amount of transaction blocking)

# Single-Thread Access

ObjectStore implements thread safety using a global *mutex* and a technique known as *mapaside*.

## Use of Global Mutex

Most access to ObjectStore API is currently serialized with one global mutex. The only exception to this is within the collection subsystem, which selectively protects a subset of the collection API. The one mutex lock protects all **libos** entry points and some of the collection entry points (such as queries). It does not protect all the collection entry points. This allows multiple threads to manipulate separate collections without blocking one another.

### Implications when using ObjectStore collections

This means that you must take care to ensure that these operations do not interfere with one another when two or more threads manipulate a single collection. You might choose to prevent the interference by using an application mutex. (Therefore you are not necessarily thread safe if two threads are operating on the same collection that has already been mapped into memory.)

ObjectStore Release 5.1 is organized to use independent mutexes for different subsystems, but currently implements one mutex. This mutex serializes both implicit access (using a page fault) and explicit access (using an API call).

## Mapaside Technique

ObjectStore's unique memory mapping architecture creates a condition that cannot be protected by a simple mutex. In order to handle such a condition ObjectStore uses the technique called *mapaside*.

### Example: mapaside

Consider an example in which two independent threads dereference a pointer to a nonmapped page **X**. If Thread 1 dereferences the pointer first, it causes a page fault on page **X**. This page fault acquires the global mutex, then fetches, maps, and relocates page **X** into the client's address space. During relocation this page must be writable so ObjectStore can relocate the page. A problem occurs if Thread 2 dereferences a pointer to page **X** during this relocation step. Since the page is writable ObjectStore will never be notified (no page fault occurs); thus, the global mutex is not checked.

### How mapaside works

Since Thread 2 could access page **X** while page **X** is in an inconsistent state, ObjectStore must

- Map the page into an intermediate address
- Perform the relocation
- After relocation, remap the page into its real location

### Platform-specific considerations

The cost of mapaside can vary widely across platforms. It depends upon the cost of the extra **mmap** calls (two additional **mmap**s) or — in a non-file-based case (for example, HP–UX device driver) — the extra **memmove**.

# Transactions

Independent of thread safety, ObjectStore Release 5.1 has a restriction that a process can have only one top-level transaction opened per process. This means that you must choose one of the following models:

- Only one thread can be in a transaction at a time (locally scoped transaction).
- Several threads can share a single transaction (globally scoped transaction).

ObjectStore provides locally scoped transactions for two reasons:

1 Single thread per transaction supports lexical transactions that can restart in case of deadlock. This means that only one thread can be accessing persistent data at a time. Since this is the case, you would not encounter a situation similar to the mapaside example; therefore, mapaside is unnecessary and can be turned off for locally scoped transactions.

2 Lexical transactions implement restart by unwinding their execution stacks and retrying the transaction. Threads do not share execution stacks, so lexical transactions need to be locally scoped.

Dynamic transactions can be scoped either locally or globally. Locally scoped transactions are serialized within a process. Globally scoped transactions must be managed by the application. The application needs to guarantee that no thread is accessing the database during a commit or abort.

## Optimizing Transactions in Threaded Environments

In a threaded environment, optimize ObjectStore by using these techniques when appropriate for your application:

- Restrict the use of locally scoped transactions. Only use locally scoped transactions when
  - The threads of an application do a lot of work outside a transaction and only a little work within a transaction.
  - Deadlock handling is critical and would be difficult to implement without locally scoped transactions.

- Keep dynamically scoped transactions open for as long as possible. The more times you access a page **X** during a transaction, the less the relative cost of relocating that page (using mapaside). Additionally, there is less time spent in blocking by the mutex and less time spent starting and committing transactions.
- Reduce the use of explicit ObjectStore API. This is unlikely to be an issue since ObjectStore API is not explicitly called very often.
- The Collection APIs are an exception to the rule because they only block at query create and execute time. For unoptimized queries it would be better to iterate than to query the collection in a multithreaded application.

# Multiple-Threaded Application Models

Two major models for multithreaded applications are

- Multiple processing threads to accomplish a single task
- Separate concurrent processing threads working on disjoint tasks, so that the views are independent and self-consistent

The first model is ideally suited for ObjectStore global transactions. The second model is very typical in application and data servers. For example, the second model is particularly suitable for Internet and intranet applications. These applications are currently very server-centric, where the network browser (front-end GUI) sends commands to the server application for processing by means of the common graphical interface (CGI).

Logically, each simple client application, also known as a *thin client*, interacts with an application server in independent work units. When planning for read-only operations, sharing transactions works well, because there is no risk of interference between such operations. However, when considering update, or write, operations, sharing transactions does not work because of the potential for

- Conflicting updates.
- Data perceived in different states.
- Read-only operations might see intermediate results from write operations.

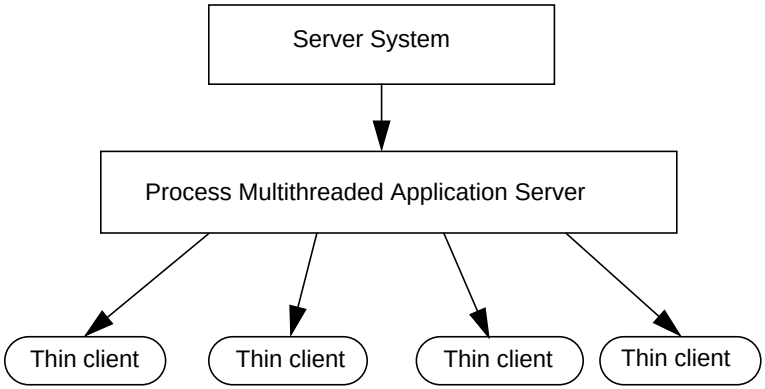
The key is to isolate the actions of writers from read-only operations. Methods of achieving the independence can rely on varied server application architectures. Several alternatives are described in the paragraphs that follow.

## One Multithreaded Process

### Simple single process architecture

The simplest process architecture would be a single process that supports all clients. Within this process you want the ability to handle multiple concurrent read-only units of work within a global transaction. You can provide a transaction manager that is responsible for serializing write transactions (which can be local transactions) and coordinating the read transaction boundaries in relationship to the write transactions.

The diagram that follows depicts such an arrangement. The *thin client* refers to a simple client application operation.




### Advantages to this architecture

This architecture has several advantages:

- Easy to write the interface (only one process to communicate with).
- Optimized for read transactions. This is an excellent design for applications that need to support a large proportion of readers.
- Easy to implement.

### Disadvantages to this architecture

The disadvantages of this design are that

- Each write operation gets a transaction, increasing the overhead.
- Readers are blocked by writers.

### Complex single process architecture

A more complicated variation for a single process would allow multiple client write transactions (units of work) per single global (or local) transaction. With this architecture you allow multiple units of work to be carried out regardless of reader or writer and return success before transaction commit. This design is more difficult since you must log input events so that they can be replayed later in case of transaction failure.

### Advantages to this architecture

The advantages of this architecture are

- Batching write transactions cuts down on transaction overhead and latency.
- Replaying from the log benefits debugging and recovery.

Disadvantages to this architecture

This architecture has several disadvantages:

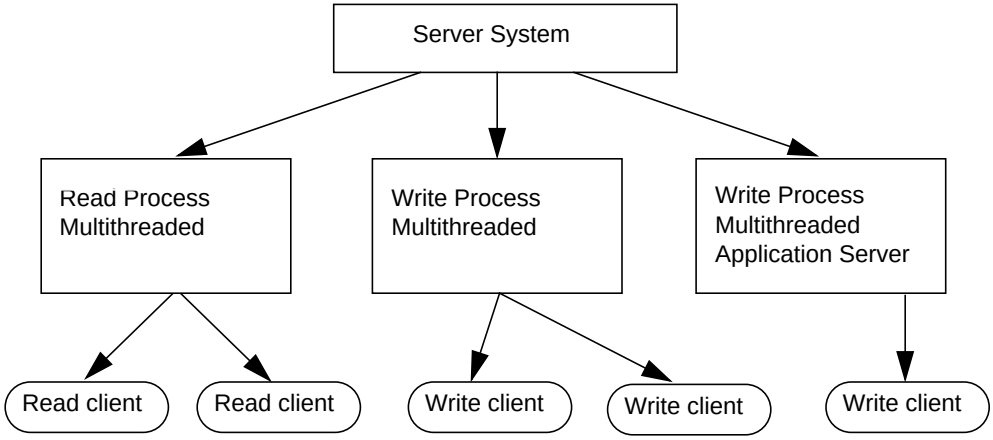
- If implementing global concurrent units of work, you must implement application logic to control visibility and serialization.
- It is hard to implement (logging might not be trivial).

## Separate Read/Write Multithreaded Processes

Another simple approach to properly handling read and write units of work is to have separate processes for readers and writers. The reader process (or processes) use global transactions and keep the transaction *open* for multiple units of work. The writer process (or processes) use local transactions and commit after each unit of work.

The writer process

Ideally, a work manager should be made responsible for selecting an appropriate writer process (from a pool of writers) for a unit of work based upon selection criteria.




The selection criteria can be

- Client ID
- Unit of work
- Available Server process

The reader process

This type of process architecture is very easy to build using *ObjectForms*. By using a separate service name for update and read units of work, you can easily direct which server the client

application communicates with. For the read-only service, you can set up the service to use multiple threads. For the write (update) service you can set up the service to use multiple processes.

Since you have a process dedicated to read-only access you can configure this process to open the database in an MVCC read-only mode and coordinate the transaction boundaries using notification from the writers. This allows you to

- Read a consistent view of the database
- Not block the writer because of a database lock
- Refresh your view when the database is updated or as desired

# Selecting the Right Application Design

In order to build the proper server application with ObjectStore or another database with this challenge, you must balance transaction boundaries, process model, and thread model with application requirements and development complexity.

# Chapter 4
# Advanced Collections

This chapter is intended to augment the information contained in Chapter 5, Collections, of the *ObjectStore C++ API User Guide*. The information contained here is organized in the following manner:

# Advanced Collections Overview

A collection is an object that serves to group together other objects. It provides a convenient means of storing and manipulating groups of objects, supporting operations for inserting, removing, and retrieving elements. Collections also support set-theory operations such as *intersection* and set-theory comparisons such as *subset*.

In addition, collections form the basis of the ObjectStore query facility, which allows you to select those elements of a collection that satisfy a specified condition. Queries with simple conditions are discussed in this chapter. Queries with complex conditions are described in Chapter 5, Queries and Indexes, on page 125.

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

# Using Paths in Navigation

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

- The **os_Cursor** class is discussed in Using Cursors for Navigation in Chapter 5 of the *ObjectStore C++ API User Guide*. See also **os_Cursor** in the *ObjectStore Collections C++ API Reference*.
- The **os_coll_range** class is described in Using Ranges in Navigation on page 76. See also **os_coll_range** in the *ObjectStore Collections C++ API Reference*.

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

## Paths

A *path*, an instance of **os_index_path**, represents a navigational path starting from the elements of a collection. Paths are used to specify index keys, and to specify a cursor's associated order. See Creating Paths on page 65, and also Paths and Member Functions on page 68.

Paths are also used in conjunction with ranges to specify a restriction on the elements a cursor visits. See Using Ranges in Navigation on page 76 for information on ranges.

# Creating Paths

Paths are used to specify traversal order (see Controlling Traversal Order on page 73) and index keys (see Indexes and Query Optimization on page 140).

## Simple Paths

Using the simplest kind of path, you can base an index key or iteration order on the value of some data member or simple member function. For example, you can iterate through a set of parts in order of the part's part numbers (parts with lower numbers precede parts with higher numbers). The member is specified with an instance of the class **os_index_path**, which designates a *member access path*.

To iterate in order of part numbers, first create a path with **os_index_path::create()**, which returns an **os_index_path&**:

```
os_index_path::create("part*", "part_number", db1)
```

The object created designates the path from part pointers to their part numbers. Here, **part*** is the path's *type string*, which names the element type of collections whose elements can serve as path starting points. **part_number** is a *path string* indicating the data member itself. The argument **db1** is a database whose schema contains the definition of the class **part**.

Both type strings and path strings can contain white space around tokens.

The instance of **os_index_path** generated by the call to **create()** is heap-allocated. When you no longer need it, you should deallocate it with the following static member of **os_index_path**:

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

## Multiple Member Paths

Sometimes path expressions specify not just a single member, but a navigational path involving multiple member accesses. For example, to base iteration order on the **emp_id** of a part's **responsible_engineer**, you create the following path:

```
os_index_path::create(
    "part*", "responsible_engineer->emp_id",
```

```
    db1
) ;
```

### Examples of path expressions

A path applied to a pointer to an employee, returning the employee's name, is specified by the path expression

```
os_index_path::create("employee*", "name", db1)
```

A path applied to a pointer to an employee, returning the employee's manager, is specified by

```
os_index_path::create(
    "employee*", "department–>manager",
    db1
)
```

A path applied to a pointer to an employee, returning the name of each of the employee's supervisees, is specified by

```
os_index_path::create(
    "employee*", "supervisees[]–>name",
    db1
)
```

Brackets (**[]**) in a path indicate that the path goes through each element of the indicated collection. The path mapping follows the remainder of the specified path for each element. Thus, application of a path mapping can result in many values.

In this last example, the path maps a single **employee*** to many names, the names of the employee's supervisees. If you use such a path to specify a key for an index, then the index would handle lookup of an employee by the name of any one of the employee's supervisees.

But paths with brackets cannot be used to specify iteration order. Iteration in order of names of supervisees is not well defined, because there is more than one supervisee. In contrast, iterating by name of supervisor, for example, is well defined, if each employee has exactly one supervisor.

Another important point is that if an index path contains a path through a collection. This collection either has to be parameterized or you must cast it to the appropriate type of parameterized collection in the index path string. This is necessary so thaat the index path parser can determine if, for example, name is a valid data member of the type of element in the supervisees collection. For example,

```
(os_Collection<employee*>&)supervisees[]->name
```

## Rank and Hash Functions

Paths that end in a class, or that end in a floating numerical type, must have associated rank and / or hash functions. And you must register these functions by calling **os_index_key()**. See The os_ index_key() Macro on page 92.

# Paths and Member Functions

Member functions called in path strings are subject to certain restrictions and prerequisites, as described in this section.

## Restrictions

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

- The return type can be any type but if it is a user-defined type, the rank / hash functions for the type must be defined.
- The function must take no arguments.

For example, consider the following class:

```
class person {
    private:
        unsigned int age;
        person* sibling;

    public:
        unsigned int get_age();

    person* get_sibling();
    unsigned int get_age_n_years_ago(unsigned int n);
}
```

The functions **get_age()** and **get_sibling()** can be referenced in a query or path string. **get_age_n_years_ago()** violates the second of the preceding restrictions.

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

## Prerequisites

Applications that use a member function in a query or path string must do four things:

1 Define an **os_backptr**-valued data member in the class that defines the member function. This data member must appear before any query functions in the class definitions.

2 Call the macro **os_query_function()**.

3 Call the macro **os_query_function_body()**.

4 Call the macro **OS_MARK_QUERY_FUNCTION()**.

You can name the **os_backptr** member anything you want. In addition, you can use the same **os_backptr** member for indexable data members and member functions; a class never needs to define more than one **os_backptr**-valued member.

## The os_query_function() Macro

A call to **os_query_function()** has the form

Form of the call

**os_query_function(***class***,***func***,***return_type***)**

where

- *class* is the name of the class defining the member function.
- *func* is the name of the member function itself.
- *return_type* names the type of value returned by the member function.

The **os_query_function()** macro should be invoked at module level in a header file (for example, the file containing the definition of the class that declares the member function). No white space should appear in the argument list.

## The os_query_function_returning_ref() Macro

The application that uses this member function, returning reference, in a query must call **os_query_function_returning_ref()**.

Form of the call

A call to this macro has the form

**os_query_function_returning_ref(***class***,***func***,***return_type***)**

where

- *class* is the name of the class defining the member function.
- *func* is the name of the member function itself.

- *return_type* names the type of value returned by the member function. The way to use this is to pass just *return_type*, not *return_type*&, to the macro *return_type* arguments.

Use of functions returning references in query strings

In query and index path strings, functions returning references should be treated as if they returned pointers. For example, queries over the function

```
Name& get_name();
```

Should be written as if the function signature were

```
Name* get_name();
```

That is,

```
*get_name() == *(Name*) Freevar
```

An index path to support this query would be

```
*get_name()
```

## The os_query_function_body() Macro

A call to **os_query_function_body()** has the form

Form of the call

```
os_query_function_body(class,func,return_type,bpname)
```

where

- *class* is the name of the class that defines the member function.
- *func* is the name of the member function itself.
- *return_type* names the type of value returned by the member function.
- *bpname* is the name of the **os_backptr**-valued member of **class**.

The **os_query_function_body()** macro should be invoked at module level in a source file (for example, the file containing the definition of the member function). No white space should appear in the argument list.

## The OS_MARK_QUERY_FUNCTION() Macro

A call to **OS_MARK_QUERY_FUNCTION()** has the form

Form of the call

```
OS_MARK_QUERY_FUNCTION(class,func)
```

where

- *class* is the name of the class that defines the member function.

- *func* is the name of the member function itself.

The **OS_MARK_QUERY_FUNCTION()** macro should be invoked along with the **OS_MARK_SCHEMA_TYPE()** macros for an application's schema, that is, in the schema source file, inside the dummy function containing the calls to **OS_MARK_SCHEMA_TYPE()**. No white space should appear in the argument list of **OS_MARK_QUERY_FUNCTION()**.

## The os_query_function_body_returning_ref() Macro

This macro enables users to register a query function that returns a reference. The application that uses this member function in a query must call **os_query_function_body_returning_ref()**:

### Form of the call

```
os_query_function_body_returning_ref(
        class,func,return_type,bpname
)
```

where

- *class* is the name of the class defining the member function.
- *func* is the name of the member function itself.
- *return_type* names the type of value returned by the member function. The way to use this is to pass just *return_type*, not *return_type*&, to the *return_type* arguments of the macro.
- *bpname* is the name of the **os_backptr**-valued member of **class**.

### Use of functions returning references in query strings

In query and index path strings, functions returning references should be treated as if they returned pointers. For example, queries over the function

```
Name& get_name();
```

Should be written as if the function signature were

```
Name* get_name()
```

That is,

```
*get_name() == *(Name*)Freevar
```

An index path to support this query would be

```
*get_name()
```

## Path String Syntax Extension

Given a path string that specifies a path ending in pointers to objects:

*path-string*

If the last component of the path string is a member function name, you can construct a path string to specify a path ending in those objects themselves, using ***** (asterisk) as follows:

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

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

Consider for example the path

```
os_index_path::create(
    "rectangle*",
    "get_location()",
    db
) ;
```

If the function **rectangle::get_location()** returns a pointer to a **coord** object, this path ends in pointers to **coord** objects. So you can also construct a path that ends in **coord** objects themselves, rather than pointers:

```
os_index_path::create(
    "rectangle*",
    "*get_location()",
    db
) ;
```

## Index Maintenance

To maintain indexes keyed by paths containing member function calls, use **os_backptr::make_link()** and **os_backptr::break_link()**. See User-Controlled Index Maintenance with an os_backptr on page 156.

# Controlling Traversal Order

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

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

## Rank-Function-Based Traversal

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

If you create a cursor using this constructor, which takes the name of the element type as argument, then iteration using that cursor will follow the order determined by the element type's rank function. See Supplying Rank and Hash Functions on page 92.

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

Rank-function-based cursors are update insensitive. See Performing Collection Updates During Traversal on page 81.

## Address Order Traversal

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

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

If you will dereference each collection element as you retrieve it, and the objects pointed to by collection elements will not all fit in

the client cache at once, this order can dramatically reduce paging overhead.

An order-by-address cursor is update insensitive.

## Path-Based Traversal

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

If you create a cursor using this constructor, which takes a reference to an **os_index_path** as an argument, then iteration using that cursor will follow the order determined by the path. See Creating Paths on page 65.

### Multiple member paths

In the case of multiple member paths (see page 65), the traversal will first visit all elements that have a complete path, and then visit the other elements in order of decreasing path length. That is, given the path **boss->boss->boss->name**, the cursor will first visit those collection elements that actually have a **boss->boss->boss** in name order, before visiting any that do not.

### Reuse of an **os_index_path**

Note that, once you have created an **os_index_path**, you can reuse it (for example, to specify the same order for another iteration, or to specify the key of an index). There is no need to create a separate index path each time you specify an iteration order or index.

### **char*** paths

Paths whose values are **char*** are treated specially by the iteration facility. An ordered iteration based on a **char***-valued member does not iterate in order of addresses (the values, strictly speaking, of the data member), but rather the iteration proceeds in the order of the strings pointed to. The order is determined by the function **strcmp()**.

### Example

Here is a code fragment demonstrating iteration by **responsible_engineer**'s **emp_id**:

```
os_index_path &a_path = os_index_path::create(
    "part*",
    "responsible_engineer–>emp_id",
    db1
) ;
```

```
os_Cursor<part*> c(a_set, a_path) ;
part *p = 0 ;
for (p = c.first(); p; p = c.next())
        printf("%d", e–>emp_id) ;
```

If an index on the path is present, it is used to make the traversal more efficient.

If an index on the iteration path is not present, the cursor constructor copies the collection elements into a separate structure, applies the path to each element, copies the terminal key into that structure, and then sorts it according to the key rank. Care is taken to sort the structure by address whenever the path interpretation calls for dereferencing a pointer, in order to improve paging behavior. Cursor creation time depends on the length of the path, the size of the collection, and the complexity of the rank function.

When you are performing path-based traversal over some collection, if you update a data member on which the ordering is based, you are effectively *removing and then reinserting* the element you changed. In other words, when you update such a data member for an element of a collection, you also update the collection itself. Therefore, for an ordered iteration with such updates, the collection's behavior specification must include **os_collection::maintain_cursors** and the cursor must be created with **os_cursor::safe**. See Performing Collection Updates During Traversal on page 81.

Keep in mind some representations of collections never allow **maintain_cursors** behavior. These are unordered types of collections.

# Using Ranges in Navigation

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

- The **os_Cursor** class is discussed in Using Cursors for Navigation in Chapter 5 of the *ObjectStore C++ API User Guide*. See also **os_Cursor** in the *ObjectStore Collections C++ API Reference*.
- The **os_index_path** class is described in Using Paths in Navigation on page 64. See also **os_index_path** in the *ObjectStore Collections C++ API Reference*.

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

## Ranges

A range, an instance of **os_coll_range**, represents a range of values. A range can be used in conjunction with a path to restrict the elements visited by a cursor. See Specifying Collection Ranges on page 77 and Restricting the Elements Visited in a Traversal on page 80.

You can also use an **os_coll_range** to retrieve from a dictionary the elements whose key falls within a specified range. See Dictionaries in Chapter 5 of the *ObjectStore C++ API User Guide*.

Both these uses of ranges provide a way of performing inexpensive, simple queries, without some of the overhead associated with using **query()**, **query_pick()**, or **exists()**. See Chapter 5, Queries and Indexes, on page 125, for more detailed information about performing queries.

# Specifying Collection Ranges

You can create an object that represents a range of values with the class **os_coll_range**. An instance of this class can be used as argument to the **os_Cursor** constructor to create a restricted cursor, or as argument to **os_Dictionary::pick()**.

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

- Overloadings that specify both a lower and upper bound on a range of values (as in “all values greater than 4 and less than or equal to 7”)
- Overloadings that specify just a lower bound or just an upper bound (as in “all values less than or equal to 7”)

In each of these two groups, there is one overloading for each C++ fundamental type of value, and one for the type **void***. To specify a range for any type of pointer value, use a **void*** overloading.

## Ranges with Only One Bound

Here are the overloadings for **os_coll_range()** that specify only an upper or lower bound.

```
os_coll_range(int rel_op, int value) ;
os_coll_range(int rel_op, unsigned int value) ;
os_coll_range(int rel_op, short value) ;
os_coll_range(int rel_op, unsigned short value) ;
os_coll_range(int rel_op, char value) ;
os_coll_range(int rel_op, unsigned char value) ;
os_coll_range(int rel_op, long value) ;
os_coll_range(int rel_op, unsigned long value) ;
os_coll_range(int rel_op, float value) ;
os_coll_range(int rel_op, double value) ;
os_coll_range(int rel_op, const void* value) ;
```

### Enumerators for the **rel_op** argument

The argument **rel_op** should be one of the following enumerators:

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

A collection range created with one of these functions is satisfied by all values that bear the relation **rel_op** to **value**. When the value type is **char***, these operators are defined in terms of **strcmp()**.

So, for example,

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

is satisfied by all values less than or equal to **7**, and

```
os_coll_range( os_collection::EQ, 7)
```

is satisfied only by the value **7**.

```
os_coll_range( os_collection::LE, "foo")
```

is satisfied by any **char*** value, **s**, such that **strcmp(s,"foo")** is less than or equal to **0**.

```
os_coll_range( os_collection::EQ, "foo")
```

is satisfied by any **char*** value, **s**, such that **strcmp(s, "foo")** is **0**.

## Ranges with Both an Upper and Lower Bound

Here are the overloadings for **os_coll_range()** that specify both an upper and lower bound.

```
os_coll_range(int rel_op1, int value1, int rel_op2, int value2) ;

os_coll_range(int rel_op1, unsigned int value1, int rel_op2,
    unsigned int value2) ;

os_coll_range(int rel_op1, short value1, int rel_op2,
    short value2) ;

os_coll_range(int rel_op1, char value1, int rel_op2, char value2) ;

os_coll_range(int rel_op1, unsigned char value1, int rel_op2,
    unsigned char value2) ;

os_coll_range(int rel_op1, long value1, int rel_op2, long value2) ;

os_coll_range(int rel_op1, unsigned long value1, int rel_op2,
    unsigned long value2) ;

os_coll_range(int rel_op1, float value1, int rel_op2, float value2) ;

os_coll_range(int rel_op1, double value1, int rel_op2,
    double value2) ;

os_coll_range(int rel_op1, const void *value1, int rel_op2,
    const void *value2) ;
```

Constructs an **os_coll_range** satisfied by all values that both bear the relation **rel_op1** to **value1** and bear the relation **rel_op2** to

**value2**. The arguments **rel_op** and **rel_op2** should be one of the following enumerators:

Enumerators for **rel_op** and **rel_op2** arguments

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

When the value type is **char***, these relations are defined in terms of **strcmp()**. So, for example:

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

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

Do not specify the null range, for example:

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

Discontinuous ranges

Do not specify a discontinuous range, for example:

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

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

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

# Restricting the Elements Visited in a Traversal

A special overloading of the **os_Cursor** constructor allows you to create a cursor for including in traversals only those collection elements that satisfy a specified restriction. Using such a cursor allows you to perform simple queries that are less expensive than queries performed with **os_collection::query()**.

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

An element satisfies the cursor's restriction if the result of applying **path** to the element satisfies **range** (see Specifying Collection Ranges on page 77). The order of iteration is arbitrary.

See also **os_Cursor** in Chapter 2 of the *ObjectStore C++ API User Guide*.

## Dictionaries

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

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

An element satisfies this cursor's restriction if its key satisfies **range**. If the dictionary's *key type* is a class, you must supply rank and hash functions for the class. See Supplying Rank and Hash Functions on page 92.

## Duplicates

With a restricted cursor, a traversal visits only one occurrence of each qualified element.

# Performing Collection Updates During Traversal

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

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

A safe cursor at a given point in a traversal visits any elements inserted later in the traversal order, and does not visit any elements that are later in the traversal order that are removed.

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

## Update-Insensitive Cursors

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

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

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

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

- Rank-function-based cursors. See Rank-Function-Based Traversal on page 73.
- **order_by_address** cursors. See Address Order Traversal on page 73.

## Safe Cursors

To traverse a collection with a safe cursor, you must specify **maintain_cursors** when you create the collection (or use **change_behavior()** — see Changing Collection Behavior with change_behavior() on page 98 — and you must pass **os_collection::safe** to the cursor constructor.

### Disadvantages of safe cursors

Safe cursors have some drawbacks that update-insensitive cursors do not:

- Updates to collections with safe cursors are slower. For each collection in a given segment that has **maintain_cursors** behavior, there is an entry in a table mapping collections to their safe cursors. This table is stored in the same segment as the collections. An update to one of the collections requires a lookup in the table. Each cursor associated with the collection is checked and adjusted if necessary.
- Index maintenance for collections with safe cursors is slower. Whenever index maintenance is performed on an object in an indexed collection that has **maintain_cursors** behavior, the safe cursor table also has to be visited (because there might be safe ordered cursors that are pointing to the indexes).
- Safe cursors are not supported for ObjectStore dictionaries. If you try to create a dictionary with **maintain_cursors** behavior, you will receive an exception.

### Using safe cursors to implement recursion

One advantage of safe cursors is that they can be used to implement recursion without the use of recursive function calls.

Consider, for example, the code below:

```
os_database *db1 ;
. . .
os_Collection<part*> &result_set =
    os_Collection<part*>::create(db1) ;
part *a_part, *p ;

. . .

result_set |= a_part ;
os_Cursor<part*> c(result_set) ;
for ( p = c.first() ; p ; p = c.next() )
    result_set |= p->children ;     /* UNSAFE!! */
```

Here, a new (empty) set, **result_set**, is created, and **a_part** is added to it. The **for** loop then iterates over the elements of **result_set**, adding the children of each element visited to **result_set** itself. The first element visited is **a_part**. But after that, the results of the iteration are undefined.

You can specify that you want a collection to support recursive queries by including **maintain_cursors** in the behavior specification. And you can specify that you want a new cursor to

be **safe** by including the enumerator **os_cursor::safe** as the second argument to the constructor for **os_Cursor**:

```
os_Collection<part*> &result_set =
    os_Collection<part*>::create(
        db1,
        os_collection::maintain_cursors
    ) ;
. . .
os_Cursor<part*> c(result_set, os_cursor::safe) ;
```

Below is an example that builds a set of all the descendents of a given part (that is, a set containing the part's children, its children's children, and so on). It is just like the first example in this section, except that the collection is to use **maintain_cursors**, and a safe cursor is created.

```
os_Collection<part*> &result_set =
    os_Collection<part*>::create(
        db1,
        os_collection::maintain_cursors
    ) ;

part *a_part, *p ;
. . .
result_set |= a_part ;

os_Cursor<part*> c(result_set, os_cursor::safe) ;

for (p = c.first(); p; p = c.next())
    result_set |= p->children ;     /* union of two sets */
```

Every child added to **result_set** is visited later in the iteration, and *its* children are added. By the end of the iteration, all the descendents are in the set.

### Example: recursive query

Below is another example of a recursive query. This one finds the *primitive* descendents of a given part, those descendents that themselves have no children:

```
os_Collection<part*> &result_set =
    os_Collection<part*>::create(
        db1,
        os_collection::maintain_cursors
    ) ;

part *a_part, *p ;
. . .
result_set |= a_part ;

os_Cursor<part*> c(result_set, os_cursor::safe) ;

for (p = c.first(); p; p = c.next())
```

```
if (p->children.size()) {
    result -= p ;    /* remove if not primitive */
    result_set |= p->children ;     /* union of two sets */
}
```

Here, each child that itself has children is removed, and the child's children are added to the set. These children just added are visited later in the iteration so that, if they have children, they can be removed and their children added. By the end of the iteration, only the primitive descendents remain in the set.

As mentioned above, recursive queries always work for unordered iteration, but performing recursive queries using ordered iteration requires more care. A value inserted during an ordered iteration is visited only if it is inserted *later*, in the order of iteration, than the current iteration position.

## Ordered, Safe Traversal

If you want to create a cursor that is both ordered and safe, supply the path argument *before* the enumerator **os_cursor::safe**:

```
os_Cursor<part*> c(a_collection, a_path, os_cursor::safe);
```

You must perform index maintenance for any data member or member function in the path used to specify the traversal order, if during iteration you update a data member controlling iteration order. See Performing or Enabling Index Maintenance on page 148.

### Caution about infinite loops

It is important to realize that there are certain dangers associated with performing updates, within an iteration, to a data member controlling iteration order. Consider, for example, the following loop, which iterates through a set of employees, giving some a raise:

```
os_Set<part*> &employees = ... ;

os_index_path &emp_path =
        os_index_path::create("employee*", "salary", db1);

os_Cursor<part*> c(
    employees,
    emp_path,
    os_cursor::safe
);
employee *e = 0;

for( e = c.first() ; e ; e = c.next() )
```

```
if ( e–>widgets_sold > 100000 )
   e–>salary *= 1.2 ;
```

Because iteration is by increasing salary, any employee who gets a raise is moved ahead in the iteration order, and so is visited again. If anyone gets a raise, the loop will be infinite. The proper approach is to use an iteration order unaffected by the updates.

# Retrieving Uniquely Specified Collection Elements

### The **retrieve()** function

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

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

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

### The **only()** function

You can retrieve the only element of a collection with

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

If the collection has more than one element, err_coll_not_singleton is signaled. If the collection is empty, err_coll_empty is signaled, unless the collection's behavior includes **os_collection::pick_from_empty_returns_null**, in which case **0** is returned.

## Ordered Collections

For collections with **maintain_order** behavior, you can retrieve the element with a specified numerical position with this function:

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

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

### The **retrieve_first()** function

You can retrieve a collection's first element with

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

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

For collections with **allow_nulls** behavior, you can use this function instead:

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

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

### The **retrieve_last()** function

To retrieve a collection's last element, use

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

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

For collections with **allow_nulls** behavior, you can use this function instead:

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

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

# Selecting Individual Collection Elements with **pick()**

The function **os_collection::pick()** can be used to perform simple queries. It provides a relatively inexpensive alternative to **os_collection::query_pick()** and **os_collection::exists()**. With the following overloading, you can retrieve a collection element such that the result of applying **path** (see Creating Paths on page 65) to the element is a value that satisfies **range** (see Specifying Collection Ranges on page 77):

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

Example: **pick()**

For example:

```
const os_index_path &id_path =
    os_index_path::create("employee*", "id", db1) ;
os_coll_range range_eq_1138(EQ, 1138) ;
. . .
employee *e = a_coll.pick(id_path, range_eq_1138) ;
```

assigns to **e** an employee in **a_coll** whose **id** equals **1138**.

If there is more than one such element, an arbitrary one is picked and returned. If there is no such element, err_coll_empty is signaled, unless the collection has behavior **os_collection::pick_from_empty_returns_null**, in which case **0** is returned.

## Dictionaries

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

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

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

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

For dictionaries, you can also retrieve an element whose key satisfies a specified collection range (see Specifying Collection Ranges on page 77) with

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

Example: dictionary **pick()**

For example:

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

returns an element of **a_dictionary** whose key is greater than or equal to **100**.

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

If the dictionary's key type is a class, you must supply rank and hash functions for the class (see Supplying Rank and Hash Functions on page 92).

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

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

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

## Picking an Arbitrary Element

You can retrieve an arbitrary collection element with

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

If the collection is empty and has **pick_from_empty_returns_null** behavior, **0** is returned. If the collection is empty and does not have **pick_from_empty_returns_null** behavior, err_coll_empty is signaled.

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

### Example: retrieving an arbitrary collection element

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

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

# Consolidating Duplicates with **operator =()**

You can use the assignment operator **os_Collection::operator =()** (see Copying, Combining, and Comparing Collections in Chapter 5 of the *ObjectStore C++ API User Guide*) to consolidate duplicates in a bag or other collection. Do this by assigning the collection with duplicates to an empty collection that does not allow (but does not signal) duplicates.

Example: consolidating duplicates

```
os_database *db1 ;
part *a_part, *p ;
employee *e ;
. . .

os_Collection<employee*> &emp_bag =
    os_Collection<employee*>::create(
        db1,
        os_collection::allow_duplicates
    ) ;

os_Collection<employee*> &emp_set =
    os_Collection<employee*>::create(db1) ;
. . .

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

emp_set = emp_bag ;     /* consolidate duplicates */

os_Cursor<employee*> c(emp_set) ;

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

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

# Supplying Rank and Hash Functions

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

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

You must also supply a rank function if you use such a path to specify a key for an ordered index (see Index Options on page 144). For unordered indexes keyed by such paths, you must supply both a rank and a hash function (the rank function is used to resolve hashing collisions). ObjectStore uses these utilities to maintain proper information on index paths that end in the class.

## The os_index_key() Macro

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

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

For example:

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

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

## Rank Functions

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

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

- **NaN** and **inf** must be handled specially. For example, the representation of **NaN** is not unique. In the rank function, test for these values before doing anything else. You might want **NaN** to rank below any other value.
- For the purpose of ranking, comparisons should be precise. For example, the rank function should consider **x** and **y** to be equal if **x == y** but not if **abs(x – y) < e** (for some small value of **e**) as long as **>** and **<** also check for equality using **e**. Using imprecise comparisons can lead to corrupt indexes and incorrect query results.

## Hash Functions

The *hash-function* is a user-defined global function that, for each instance of **class**, returns a value, an **os_unsigned_int32**, that can be used as a key in a hash table. It takes a **const void*** argument. If you are not supplying a hash function for the class, this argument should be **0**.

## Example Use of Rank and Hash Functions

Suppose you have a collection of pointers to messages, instances of a class that you have defined. Further suppose you want to iterate through the messages in order of their dates to display each message. If a message has an indexable data member whose value is its date, you can code such an iteration this way:

```
os_collection &messages = . . . ;
message *a_msg;

os_index_path &date_path =
    os_index_path::create("message", "date_received", msg_db);

os_cursor c(messages, date_path);
for (a_msg = (message*) c.first(); a_msg; a_msg =
        (message*)c.next()) a_msg->display();
```

This assumes that dates have an order that is known to ObjectStore. This is true if dates are instances of an integer or pointer type such as **int**. But suppose that dates are instances of a user-defined class:

```
class date{
    public:
```

```
        int month;
        int day;
        int year;
};
```

In this case you must define the rank function and make it known to ObjectStore. Here is how you might define it:

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

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

If you also use unordered indexes keyed by **date**, you must supply a hash function, which might be defined this way:

```
os_unsigned_int32 date_hash(const void* x) {
    const date* d = (const date*) x;
    return ((os_unsigned_int32) (d->year) << 16) ^
            (d->month << 8) ^ d->day;
}
```

Finally, you must call **os_index_key()** before your application performs any iteration or query employing an index path ending on a data member of type **date**.

```
main() {
    . . .
    os_index_key(date,date_rank,date_hash);
    . . .
}
```

If unordered indexes are never created, the hash function is not needed, and the registration could be done as follows:

```
os_index_key(date,date_rank,0);
```

# Specifying Expected Size

Frequently, a collection has a loading phase, in which it is loaded with elements before being the subject of other kinds of manipulation such as queries and traversal. In these cases, it is desirable to create the collection with a representation appropriate to the size it will have once loading is complete. This saves on the overhead of transforming the collection's representation as it grows during the initial loading phase, and can improve locality of reference.

To presize a collection, use the **expected_size** argument to a collection **create()** operation or constructor.

# Customizing Collection Behavior

You can customize the behavior of new collections with regard to the optional properties for the collection's type. You do this by supplying a **behavior** argument to **create()**, an unsigned 32-bit integer, a bit pattern indicating the collection's properties. The bit pattern is obtained by using **|** (bitwise or) to form the bit-wise disjunction of enumerators taken from the following possibilities.

## Behavior Enumerators for Collection Subtypes

The following enumerators can be applied via the **behavior** argument to **create()** or **change_behavior()** for **os_Collection** and its subtypes. Enumerators that can be applied to the **os_Dictionary** class are listed in the following section. These enumerators are all described in Chapter 2, Collection, Query, and Index Classes, of the *ObjectStore Collections C++ API Reference*.

- **os_collection::pick_from_empty_returns_null**: Performing **pick()** on an empty result of querying the collection returns **0** rather than raising an exception.
- **os_collection::allow_nulls**: The pointer **0** is allowed as an element.
- **os_collection::maintain_cursors**: For traversals with *safe cursors* (see Performing Collection Updates During Traversal on page 81), an element inserted during a traversal will be visited later in that same traversal. An element removed during a traversal of its elements will not be visited later in that same traversal.
- **os_collection::allow_duplicates**: Duplicate elements are allowed.
- **os_collection::be_an_array**: For collections that maintain order only. With this behavior, access to the $n^{th}$ element is an **O(1)** operation.

## Behavior Enumerators for Dictionaries

The following enumerators can be applied using the **behavior** argument to **create()** for **os_Dictionary.** These enumerators are all described in Chapter 2, Collection, Query, and Index Classes, of the *ObjectStore Collections C++ API Reference*.

- **os_collection::pick_from_empty_returns_null**: Performing **pick()** on an empty dictionary returns **0** rather than raising an exception.
- **os_dictionary::signal_dup_keys**: Duplicate keys are not allowed; err_am_dup_key is signaled if an attempt is made to establish two or more elements with the same key.
- **os_dictionary::maintain_key_order**: Range lookups are supported using **pick()** or restricted cursors. See Selecting Individual Collection Elements with pick() on page 88 and Restricting the Elements Visited in a Traversal on page 80.
- **os_dictionary::dont_maintain_size**: For dictionaries that maintain key order only. With this behavior **insert()** and **remove()** do not update size information, avoiding contention in the collection header. This can significantly improve performance for large dictionaries subject to contention. The disadvantage of this behavior is that **size()** is an **O(n)** operation, requiring a scan of the whole dictionary. See os_ixonly and os_ixonly_bc on page 109.

## Required and Forbidden Behaviors

Here is a table summarizing the required and forbidden behaviors of the collection subtypes:

| *Collections Class* | *Allow Duplicates* | *Signal Duplicates* | *Maintain Order* | *Allow Nulls* | *O(1) Positional Access* | *Maintain Cursors* |
|---|---|---|---|---|---|---|
| **os_Set** | Forbidden | Off by default | Forbidden | Off by default | Forbidden | Forbidden |
| **os_Bag** | Required | Forbidden | Forbidden | Off by default | Forbidden | Forbidden |
| **os_List** | On by default | Off by default | Required | Off by default | Off by default | Off by default |
| **os_Dictionary** | Required | Forbidden | Forbidden | Required | Not applicable | Forbidden |
| **os_Array** | On by default | Off by default | Required | Required | Required | Off by default |

## Changing Collection Behavior with change_behavior()

You can change the behavior of a collection any time after its creation with the function **os_Collection::change_behavior()**. There are restrictions about what combinations of behaviors are allowed. Any illegal combination causes an exception to be signaled when **change_behavior** is called.

### Fully specifying the new behavior

You supply a bit pattern (an unsigned 32-bit integer) as argument, fully specifying the behavior the collection is to have after the change.

```
os_database *db1; . . .
os_Collection<part*> &some_parts =
    os_Collection<part*>::create(
        db1,
        os_collection::allow_nulls,
    ) ;
. . .
some_parts.change_behavior(
    os_collection::maintain_order |
    os_collection::allow_duplicates
) ;
```

This changes the collection **some_parts** from an unordered collection that allows nulls but does not allow duplicates into an ordered collection that does not allow nulls and does allow duplicates. Both before and after the change, **maintain_cursors** is off.

### Changing to default behavior

The following call changes **some_parts** into a collection with default behavior for its type, whatever its original behavior:

```
some_parts.change_behavior(0) ;
```

### Removing behavior

To remove behavior from a collection, you can conjoin (using **&**) the collection's current behavior with the bit-wise negation (**~**) of the enumerator representing the behavior to remove. You obtain a bit pattern representing the current behavior with **os_collection::get_behavior()**, which returns an **os_unsigned_int32** (a 32-bit unsigned integer).

Here is a function that changes a collection to disallow duplicates and signal duplicates, and that otherwise does not affect its original behavior:

```
f(os_collection &some_parts) {
    os_unsigned_int32 current_behavior =
```

```
        some_parts.get_behavior() ;
    some_parts.change_behavior(
        (current_behavior & ~os_collection::allow_duplicates) |
        os_collection::signal_duplicates
    ) ;
}
```

### Incompatibility between behavior and representation

Some behavior is incompatible with some of the possible collection representations. If a collection whose behavior you are changing has a user-defined representation policy, and that policy is incompatible with the new behavior, an err_illegal_arg exception is signaled. If the collection has the default representation policy, the representation can change to accommodate the new behavior.

### Automatic checking for nulls and duplicates

When you change a collection so that it no longer allows duplicate or null insertions, you might want to check to see if duplicates or nulls are already present. Such a check is performed for you if you supply the enumerator **os_collection::verify** as the second argument.

```
os_database *db1; . . .
os_Collection<part*> &some_parts =
    os_Collection<part*>::create(
            db1,
            os_collection::allow_nulls |
            os_collection::allow_duplicates |
            os_collection::maintain_order
    ) ;
. . .
some_parts->change_behavior(
    os_collection::maintain_order,
    os_collection::verify
) ;
```

This changes **some_parts** from an ordered collection that allows both nulls and duplicates into an ordered collection that allows neither nulls nor duplicates. The argument **os_collection::verify** indicates that the function should check for duplicates and nulls.

If nulls are found, err_coll_nulls is signaled. If duplicates are found, and **signal_duplicates** is on, err_coll_duplicates is signaled. If **signal_duplicates** is not on, the first among each set of duplicates is retained and trailing duplicates are silently removed.

If **os_collection::verify** is not used, the resulting collection is assumed to be free of duplicates or nulls. This could lead to application errors if this is not the case.

# Customizing Collection Representation

Each ObjectStore collection type permits a variety of representations. The collection *type* (**os_Set** or **os_Bag** or **os_List**, and so on) determines the allowable *behavior*; the collection *representation* determines the *performance* and *storage* characteristics.

## Representation Classes

The performance and storage characteristics of each representation are discussed in the following sections of this chapter:


- **os_chained_list** on page 102
- **os_dyn_bag** on page 105
- **os_dyn_hash** on page 107
- **os_ixonly and os_ixonly_bc** on page 109
- **os_ordered_ptr_hash** on page 112
- **os_packed_list** on page 114
- **os_ptr_bag** on page 116
- **os_vdyn_bag** on page 118
- **os_vdyn_hash** on page 120


A summary of these classes is provided in Summary of Representation Types on page 122, and all the classes are described in detail in Chapter 3, Representation Types, of the *ObjectStore Collections C++ API Reference*.

## Creating Collection Representation Objects

### Creating **os_rep** objects

You create an **os_rep** object with the **os_rep** constructor:

```
os_rep(os_rep_type rep_enum, os_unsigned_int32 size) ;
```

### Enumerators for the first argument of the **os_rep** constructor

The first argument to the **os_rep** constructor is one of the enumerators listed below. See **os_rep::os_rep()** in Chapter 2 of the *ObjectStore Collections C++ API Reference* for more information.

- **os_chained_list_rep**
- **os_dyn_bag_rep**
- **os_dyn_hash_rep**

- **os_ixonly_rep**
- **os_ixonly_bc_rep**
- **os_packed_list_rep**
- **os_ptr_bag_rep**
- **os_vdyn_bag_rep_os_reference**
- **os_vdyn_hash_rep_os_reference**

The second argument to the **os_rep** constructor is the size at which the transition to the next representation is to be made. Use

```
~(os_unsigned_int32)0
```

to specify no upper bound.

## Changing Collection Representation with change_rep()

You can change a collection's associated representation type at any time, using the member function **os_collection::change_rep()**.

```
temp_bag->change_rep (
    100,
    &os_rep
);
```

The first argument is the expected size, used just as it is in **create** to determine the initially active representation. The second argument is the **os_rep**.

If you specify an **os_rep** and expected size that determine an initial representation incompatible with the behavior of the collection you want to change, err_coll_illegal_arg is signaled. Moreover, if the type and size subsequently determine a representation incompatible with the collection's behavior, err_coll_illegal_arg is also signaled.

Note that changing the representation of a collection is in effect creating a new collection and copying all elements from the old to the new collection.

# os_chained_list

The class **os_chained_list** is a representation type that is optimized (in both time and space) for small to medium-sized collections. Each **os_chained_list** consists of a header and any number of blocks. The header has a **vptr**, one word of state, and up to 20 pointers. When the number of pointers in the header is exhausted, an **os_chained_list_block** is allocated, and chained to the header.

Each **os_chained_list_block** can contain up to 255 pointers. It has two or three words of overhead: one word of state information, a *previous* pointer, and possibly a *next* pointer (the first **os_chained_list_block** allocated does not have a *next* pointer until the next block is allocated). The default version of **os_chained_list** contains four pointers in the header and seven or eight pointers in its blocks.

The maximum size for **os_chained_list**s is 131070.

For more information, see **os_chained_list** in Chapter 3 in the *ObjectStore Collections C++ API Reference*.

## Controlling the Number of Pointers

When you create an **os_chained_list**, what is really allocated is an instance of a parameterized class derived from **os_chained_list**: **os_chained_list_pt<NUM_PTRS_IN_HEAD,NUM_PTRS_IN_BLOCKS>**. The default parameterization is <4,8>, but you can specify a different parameterization with the following macros:

- **OS_MARK_CHAINED_LIST_REP(ptrs_in_header,ptrs_in_blocks)**
- **OS_INSTANTIATE_CHAINED_LIST_REP(ptrs_in_header,ptrs_in_blocks)**
- **OS_INITIALIZE_CHAINED_LIST_REP(ptrs_in_header,ptrs_in_blocks**)

Use **OS_MARK_CHAINED_LIST_REP()** in the same dummy function as **OS_MARK_SCHEMA_TYPE()**.

Use **OS_INSTANTIATE_CHAINED_LIST_REP()** at file scope. It declares some static state needed by the representation.

Execute **OS_INITIALIZE_CHAINED_LIST_REP()** in a function. It registers the new parameterization with the collections library.

Include the files **<coll/chlist.hh>**, **<coll/chlistpt.hh>**, and **<coll/chlistpt.c>** if you use these macros.

In order to create a collection using a chained list with other than the default parameterization, you invoke the following static member function:

```
static os_coll_rep_descriptor*
os_chained_list_descriptor::find_rep(os_int32 ptrs_in_hdr,
    os_int32 ptrs_in_blocks);
```

If the requested parameterization has been specified with the above macros, the appropriate representation descriptor is returned. Otherwise, **0** is returned.

Note that an **os_chained_list** must have at least four pointers in the header but not more than 20 pointers.

An **os_chained_list** with a four-pointer header can change freely into any other collection representation and the reverse. However, other collection representations cannot change into **os_chained_lists** with more than four pointers in the header. A normal collection header is 24 bytes. An **os_chained_list** with more than four pointers exceeds this limit. It is possible for an **os_chained_list** with an oversized header to change into another representation (with the same or smaller size header).

## Pool Allocation of Blocks

You can request pool allocation of **os_chained_list_block**s with the environment variable **OS_COLL_POOL_ALLOC_CHLIST_BLOCKS** to the function **os_chlist_pool::configure_pool()**. In some cases this decreases the time needed for individual allocation of **os_chained_list_blocks** and increases the chance of getting good locality of reference.

Setting **OS_COLL_POOL_ALLOC_CHLIST_BLOCKS** turns on pool allocation. There is one pool per segment; each pool consists of an array of subpools. Each subpool is two pages by default.

By allocating larger subpools, you can defer the cost of allocating new subpools at the expense of potentially wasted space. To allocate larger subpools, use this function:

```
static void
os_chlist_pool::configure_pool(
        os_unsigned_int32 config_options,
        os_unsigned_int32 blks_per_subpool=2);
```

**config_options** can have one of the following values:

- **os_chlist_pool_no_pooled_allocation**
- **os_chlist_pool_allocate_blks**

The second argument, which is optional and defaults to **2**, controls the number of pages allocated per subpool.

## Mutation Checks

In order to improve performance, an **os_chained_list** does not necessarily check to see if it should change to another representation after every insert or remove operation. By default, it checks when the size is roughly a multiple of **7**. However, you can control the frequency with which it checks by invoking the static member function.

```
static void
os_chained_list_descriptor::set_reorg_check_interval(
    os_unsigned_int32 v);
```

ObjectStore sets the check interval to one less than the power of **2** that is greater than or equal to **v**. For example, in order to check on every other insert or remove, pass **1** or **2** as an argument. Passing **3** or **4** results in a check on every third operation. Passing **0** inhibits mutation. However, if the maximum size for an **os_chained_list** is reached, it will change to another representation.

## mutate_when_full Behavior

For collections whose representation is **os_chained_list**, if you specify the behavior enumerator **os_collection::chained_list_mutate_when_full**, the collection's representation will not change until it reaches the maximum size for chained lists.

# os_dyn_bag

Instances of this class are used as ObjectStore collection representations. The **os_dyn_bag** representation supports **O(1)** element lookup, which means that operations such as **contains()** and **remove()** are **O(1)** (in the number of elements). But an **os_dyn_bag** takes up somewhat more space than an **os_packed_list**.

The representation **os_dyn_bag** minimizes reorganization overhead at the expense of some extra space overhead, compared with **os_ptr_bag**. At large sizes, **os_dyn_bag** uses a structure pointing to many small hash tables that can reorganize independently.

This representation type does not support **maintain_order** or **maintain_cursors** behavior.

For sizes below 20, **os_chained_list** might be a better representation type.

For more information, see **os_dyn_bag** in Chapter 3 in the *ObjectStore Collections C++ API Reference*.

## Time Complexity

In the following table, complexities are shown in terms of collection size, represented by **n**. (These complexities reflect the nature of the computational overhead involved, not overhead due to disk I/O and network traffic.)

| | |
|---|---|
| **insert()** | **O(1)** |
| **remove()** | **O(1)** |
| **size()** | **O(1)** |
| **contains()** | **O(1)** |
| Comparisons (**<=**, **==**, and so on) | **O(n)** |
| Merges (**|**, **&**, **–**) | **O(n)** |

## Space Overhead

If

**size <= 64k**

the small-medium size data structure is used. It contains the following:

- Header (24 bytes)
- Entry for each element (eight bytes each)
- Some number of empty entries (eight bytes each)

On average, an **os_dyn_bag** at low-medium sizes is 69% full. You can estimate the average size as follows:

**Avg. total size in bytes = 24 + (size/.69) * 8**

If

**size > 64k**

the large size data structure is used. It contains the following:

- Header (24 bytes)
- Directory (60 byte header + 12 bytes per directory entry)
- Some number of small hash tables (two pages each, eight bytes per entry)

On average, each small hash table in an **os_dyn_bag** at high sizes is 70% full. You can estimate the average size as follows:

**n_entries = Avg. number of entries per small hash table = (8192/8) * .7**

**n_tables = Avg. number of small hash tables = size / n_entries**

**dir_size = Avg. directory size in bytes = 60 + (n_tables+1) * 12**

**Avg. total size in bytes = 24 bytes + dir_size + n_tables * 8192**

# os_dyn_hash

Instances of this class are used as ObjectStore collection representations. The dynamic hash representation supports **O(1)** element lookup, which means that operations such as **contains()** and **remove()** are **O(1)** (in the number of elements). But an **os_dyn_hash** takes up somewhat more space than an **os_packed_list**.

At large sizes, **os_dyn_hash** uses a structure pointing to many small hash tables that can reorganize independently.

This representation type does not support **allow_duplicates**, **maintain_order**, or **maintain_cursors** behavior.

For sizes below 20, **os_chained_list** might be a better representation type.

For more information, see **os_dyn_hash** in Chapter 3 in the *ObjectStore Collections C++ API Reference*.

## Time Complexity

In the following table, complexities are shown in terms of collection size, represented by **n**. (These complexities reflect the nature of the computational overhead involved, not overhead due to disk I/O and network traffic.)

| | |
|---|---|
| **insert()** | **O(1)** |
| **remove()** | **O(1)** |
| **size()** | **O(1)** |
| **contains()** | **O(1)** |
| Comparisons (**<=**, **==**, and so on) | **O(n)** |
| Merges (**|**, **&**, **–**) | **O(n)** |

## Space Overhead

If

**size <= 64k**

the small-medium size data structure is used. It contains the following:

- Header (24 bytes)
- Entry for each element (four bytes each)

- Some number of empty entries (four bytes each)

On average, the an **os_dyn_hash** at low-medium sizes is 69% full. You can estimate the average size as follows:

**Avg. total size in bytes = 24 + (size/.69) * 4**

If

**size > 64k**

the large size data structure is used. It contains the following:

- Header (24 bytes)
- Directory (60 byte header + 12 bytes per directory entry)
- Some number of small hash tables (two pages each, four bytes per entry)

On average, each small hash table in an **os_dyn_hash** at high sizes is 70% full. You can estimate the average size as follows:

**n_entries = Avg. number of entries per small hash table = (8192/4) * .7**

**n_tables = Avg. number of small hash tables = size / n_entries**

**dir_size = Avg. directory size in bytes = 60 + (n_tables+1) * 12**

**Acg. total size in bytes = 24 bytes + dir_size + n_tables * 8192**

# **os_ixonly** and **os_ixonly_bc**

Instances of these classes are used as ObjectStore collection representations. They are both index-only representations that support **O(1)** element lookup. Operations such as **contains()** and **remove()** are **O(1)** (in the number of elements). But they take up somewhat more space than an **os_packed_list**.

For large collections subject to contention, **os_ixonly_bc** can provide significantly better performance than **os_ixonly**. See **os_ixonly_bc**, below.

The next chapter discusses associating indexes with collections to improve the efficiency of queries. With **os_ixonly** or **os_ixonly_bc**, you can save space by telling ObjectStore to record the membership of the collection in one of its indexes, as opposed to recording the membership in both the index and the collection. In other words, you can save space by using an index as a collection's representation.

When these representation types are specified for a collection, you must add an index to it before any operations are performed on it. Additional indexes can also be added.

These representation types are incompatible with the following behaviors: **maintain_order**, **maintain_cursors**, **allow_nulls**, and **allow_duplicates**.

Note that using these representations can save on space overhead at the expense of reducing the efficiency of some collection operations. If the only time-critical collection operation is index-based element lookup, an index-only representation is likely to be beneficial.

For sizes below 20, **os_chained_list** might be a better representation type.

For more information, see **os_ixonly and os_ixonly_bc** in Chapter 3 in the *ObjectStore Collections C++ API Reference*.

## os_ixonly_bc

**os_ixonly_bc** is just like **os_ixonly**, except that **insert()** and **remove()** do not update size information, avoiding contention in

the collection header. The disadvantage of **os_ixonly_bc** is that **size()** is an **O(n)** operation, requiring a scan of the whole collection.

You can determine if a collection updates its size in this way with the following member of **os_collection**:

```
os_int32 size_is_maintained() const;
```

This function returns nonzero if the collection maintains size; it returns **0** otherwise.

The following member of **os_collection**, which returns an estimate of a collection's size, is an **O(1)** operation in the size of the collection:

```
os_unsigned_int32 size_estimate() const;
```

This function returns the size as of the last call to **os_collection::update_size()**. For collections that maintain size, the actual size is returned.

Before you add a new index to an **os_ixonly_bc** collection, call the following member of **os_collection**:

```
os_unsigned_int32 update_size();
```

If you do not, **add_index()** will work correctly, but less efficiently than if you do. This function updates the value returned by **os_collection::size_estimate()** by scanning the collection and computing the actual size.

## Time Complexity

In the following table, complexities are shown in terms of collection size, represented by **n**. (These complexities reflect the nature of the computational overhead involved, not overhead due to disk I/O and network traffic.)

| | |
|---|---|
| **insert()** | **O(1)** |
| **remove()** | **O(1)** |
| **size(), os_ixonly** | **O(1)** |
| **size(), os_ixonly_bc** | **O(n)** |
| **contains()** | **O(1)** |
| Comparisons (**<=**, **==**, and so on) | **O(n)** |
| Merges (**|**, **&**, **–**) | **O(n)** |

If there are safe cursors open on a particular collection, each insert or remove operation visits each of those cursors and adjusts them if necessary. This locks the info segment and can cause contention.

# os_ordered_ptr_hash

Instances of this class are used as ObjectStore collection representations. Unlike the other hash tables, this representation supports **maintain_order** behavior. The ordered pointer hash representation supports **O(1)** element lookup, which means that operations such as **contains()** and **remove()** are **O(1)** (in the number of elements). But an **os_ordered_ptr_hash** takes up somewhat more space than an **os_packed_list**.

This representation type does not support **be_an_array** behavior.

For sizes below 20, **os_chained_list** might be a better representation type.

For more information, see **os_ordered_ptr_hash** in Chapter 3 in the *ObjectStore Collections C++ API Reference*.

## Time Complexity

In the following table, complexities are shown in terms of collection size, represented by **n**. (These complexities reflect the nature of the computational overhead involved, not overhead due to disk I/O and network traffic).

| | |
|---|---|
| **insert()** | **O(1)** |
| Position-based **insert()** | **O(n)** |
| **remove()** | **O(1)** |
| Position-based **remove()** | **O(n)** |
| **size()** | **O(1)** |
| **contains()** | **O(1)** |
| Comparisons (**<=**, **==**, and so on) | **O(n)** |
| Merges (**|**, **&**, **–**) | **O(n)** |

If there are safe cursors open on a particular collection, each insert or remove operation visits each of those cursors and adjusts them if necessary. This locks the info segment and can cause contention.

## Space Overhead and Clustering

An ordered pointer hash has the following components:

- Header

- Entry for each element
- Some number of empty entries

The entry for a given element is likely to be on a different page from the collection header.

On average, a pointer hash is 58.3% full. You can estimate the average size of a pointer hash as follows:

```
if size <= 65535
        average total size in bytes = 56 + size * 8 / 58.3

if size > 65535
        average total size in bytes = 56 + size * 12 / 58.3
```

The minimum fill for a packed list is 46.7%, so an upper bound on collection space overhead can be calculated as follows:

```
if size <= 65535
        maximum total size in bytes = 56 + size * 8 / 46.7

if size > 65535
        maximum total size in bytes = 56 + size * 12 / 46.7
```

# os_packed_list

Instances of this class are used as ObjectStore collection representations. The packed list representation is relatively space-efficient, but element lookup is an **O(n)** operation, which means that operations such as **remove()** and **contains()** are **O(n)** (in the number of elements). If duplicates are allowed, this representation provides the fastest insertion times, but if duplicates are not allowed (requiring element lookup to check for the presence of a duplicate) **insert()** is **O(n)**.

For sizes below 20, **os_chained_list** might be a better representation type.

For more information, see **os_packed_list** in Chapter 3 in the *ObjectStore Collections C++ API Reference*.

## Time Complexity

In the following table, complexities are shown in terms of collection size, represented by **n**. (These complexities reflect the nature of the computational overhead involved, not overhead due to disk I/O and network traffic.)

| | |
|---|---|
| **insert()**, duplicates allowed | **O(1)** |
| **insert()**, duplicates not allowed | **O(n)** |
| Position-based **insert()**, no “holes” | **O(1)** |
| Position-based **insert()**, with “holes” | **O(n)** |
| **remove()** | **O(n)** |
| Position-based **remove()**, no “holes” | **O(1)** |
| Position-based **remove()**, with “holes” | **O(n)** |
| **size()** | **O(1)** |
| **contains()** | **O(n)** |
| Comparisons (**<=**, **==**, and so on) | **O($n^2$)** |
| Merges (**|**, **&**, **–**) | **O($n^2$)** |

There might be “holes” in an **os_packed_list** if any elements have been removed.

If there are safe cursors open on a particular collection, each insert or remove operation visits each of those cursors and adjusts them if necessary.

## Space Overhead and Clustering

A packed list has the following components:

- Header
- Entry for each element
- Some number of empty entries

The entry for a given element is likely to be on a different page from the collection header.

On average, a packed list is 83.3% full. You can estimate the average size of a collection as follows:

**average total size in bytes = 40 + size * 4 / 83.3**

The minimum fill for a packed list is 66.7%, so an upper bound on collection space overhead can be calculated as follows:

**maximum total size in bytes = 40 + size * 4 / 66.7**

# os_ptr_bag

Instances of this class are used as ObjectStore collection representations. The pointer hash representation supports **O(1)** element lookup, which means that operations such as **contains()** and **remove()** are **O(1)** (in the number of elements). But an **os_ptr_bag** takes up somewhat more space than an **os_packed_list**.

In addition, as an **os_ptr_bag** grows, there can be overhead during collection updates, for reorganization. In contrast, the representation **os_dyn_bag** minimizes reorganization overhead at the expense of some extra space overhead. At large sizes, **os_dyn_bag** uses a structure pointing to many small hash tables that can reorganize independently. See **os_dyn_bag** on page 105.

This representation type does not support **maintain_order** behavior.

For sizes below 20, **os_chained_list** might be a better representation type.

For more information, see **os_ptr_bag** in Chapter 3 in the *ObjectStore Collections C++ API Reference*.

## Time Complexity

In the following table, complexities are shown in terms of collection size, represented by **n**. (These complexities reflect the nature of the computational overhead involved, not overhead due to disk I/O and network traffic.)

| | |
|---|---|
| **insert()** | **O(1)** |
| **remove()** | **O(1)** |
| **size()** | **O(1)** |
| **contains()** | **O(1)** |
| Comparisons (**<=**, **==**, and so on) | **O(n)** |
| Merges (**|**, **&**, **–**) | **O(n)** |

If there are safe cursors open on a particular collection, each insert or remove operation visits each of those cursors and adjusts them if necessary.

## Space Overhead and Clustering

A pointer hash has the following components:

- Header
- Entry for each element
- Some number of empty entries
- Count slot for each entry
- Some number of empty count slots

The entry for a given element is likely to be on a different page from the collection header. In addition, the count slot for a given element is likely to be stored on a different page from both the header and the entry for the element.

On average, a pointer bag is 58.3% full. You can estimate the average size of a pointer bag as follows:

**average total size in bytes = 48 + size * 8 / 58.3**

The minimum fill for a packed list is 46.7%, so an upper bound on collection space overhead can be calculated as follows:

**maximum total size in bytes = 48 + size * 8 / 46.7**

# os_vdyn_bag

Instances of this class are used as ObjectStore collection representations. The **os_vdyn_bag** representation saves on relocation overhead and address space by recording its membership using ObjectStore references instead of pointers. It supports **O(1)** element lookup, which means that operations such as **contains()** and **remove()** are **O(1)** (in the number of elements). But an **os_vdyn_bag** takes up somewhat more space than an **os_packed_list**.

The representation **os_vdyn_bag** minimizes reorganization overhead at the expense of some extra space overhead, compared with **os_ptr_bag**. At large sizes, **os_vdyn_bag** uses a structure pointing to many small hash tables that can reorganize independently.

This representation type does not support **maintain_order** or **maintain_cursors** behavior.

For sizes below 20, **os_chained_list** might be a better representation type.

This class is parameterized, with a parameter indicating the type of ObjectStore reference to use for recording membership. Actual parameters can be **os_reference,** for general use.

For more information, see **os_vdyn_bag** in Chapter 3 in the *ObjectStore Collections C++ API Reference*.

## Time Complexity

In the following table, complexities are shown in terms of collection size, represented by **n**. (These complexities reflect the nature of the computational overhead involved, not overhead due to disk I/O and network traffic.)

| | |
|---|---|
| **insert()** | **O(1)** |
| **remove()** | **O(1)** |
| **size()** | **O(1)** |
| **contains()** | **O(1)** |
| Comparisons (**<=**, **==**, and so on) | **O(n)** |
| Merges (**|**, **&**, **–**) | **O(n)** |

## Space Overhead

For an **os_vdyn_bag** whose reference type parameter is **REF_TYPE**, if

```
size <= 64k
```

the small-medium size data structure is used. You can estimate its size as follows:

```
extra_slot = (((size / .69) % 16) ? 1 : 0)
average total size = 24 bytes (header) +
        ( ((size / .69) / 16) + extra_slot) *
        ((sizeof(REF_TYPE) * 16) + (16 * 4) + 4)
```

If

```
size > 64k
```

the large size data structure is used. You can estimate its size as follows:

```
entry_size:
os_reference: 20
os_reference_version: 28

if REF_TYPE != os_reference_protected:
        n_tables = (size / ( ((8192 / <entry-size>) * 2) * .7)
else
        n_tables = (size / ( (8192 / (<entry-size> )) * .7))
dir_size= (n_tables +1) * 12 bytes + 60
average total size = 24 bytes (header) +
        dir_size + n_tables * 8192 bytes
```

# os_vdyn_hash

Instances of this class are used as ObjectStore collection representations. The **os_vdyn_hash** representation saves on relocation overhead and address space by recording its membership using ObjectStore references instead of pointers. It supports **O(1)** element lookup, which means that operations such as **contains()** and **remove()** are **O(1)** (in the number of elements). But an **os_vdyn_hash** takes up somewhat more space than an **os_packed_list**.

At large sizes, **os_vdyn_hash** uses a structure pointing to many small hash tables that can reorganize independently.

This representation type does not support **allow_duplicates**, **maintain_order**, or **maintain_cursors** behavior.

For sizes below 20, **os_chained_list** might be a better representation type.

This class is parameterized, with a parameter indicating the type of ObjectStore reference to use for recording membership. Actual parameters can be **os_reference,** for general use.

For more information, see **os_vdyn_hash** in Chapter 3 in the *ObjectStore Collections C++ API Reference*.

## Time Complexity

In the following table, complexities are shown in terms of collection size, represented by **n**. (These complexities reflect the nature of the computational overhead involved, not overhead due to disk I/O and network traffic.)

| | |
|---|---|
| **insert()** | **O(1)** |
| **remove()** | **O(1)** |
| **size()** | **O(1)** |
| **contains()** | **O(1)** |
| Comparisons (**<=**, **==**, and so on) | **O(n)** |
| Merges (**|**, **&**, **–**) | **O(n)** |

## Space Overhead

For an **os_vdyn_bag** whose reference type parameter is **REF_TYPE**, if

```
size <= 64k
```

the small-medium size data structure is used. You can estimate its size as follows:

```
extra_slot = (((size / .69) % 16) ? 1 : 0)
average total size = 24 bytes (header) +
    ( ((size / .69) / 16) + extra_slot) *
    ((sizeof(REF_TYPE) * 16) + 4)
```

If

```
size > 64k
```

the large size data structure is used. You can estimate its size as follows:

```
entry_size:
    os_reference: 12
    os_reference_version: 20

if REF_TYPE != os_reference_protected:
        n_tables = (size / ( ((8192 / <entry-size> ) * 2) * .7))
else
        n_tables = (size / ( (8192 / (<entry-size> )) * .7))

dir_size= (n_tables +1) * 12 bytes + 60
average total size = 24 bytes (header) + dir_size +
        n_tables * 8192 bytes
```

# Summary of Representation Types

## Time Complexity Summary

In the table below, complexities are shown in terms of collection size, represented by **n**. (These complexities reflect the nature of the computational overhead involved, not overhead due to disk I/O and network traffic.)

An ordered hash table is an **os_ordered_ptr_hash**. Unordered hash tables include the following:

- **os_dyn_bag** on page 105
- **os_dyn_hash** on page 107
- **os_ptr_bag** on page 116
- **os_vdyn_bag** on page 118
- **os_vdyn_hash** on page 120

| | ***Unordered Hash Tables*** | ***Ordered Hash Tables*** | **os_packed_ list** | **os_ixonly** | **os_chained_ list** |
|---|---|---|---|---|---|
| **insert()** | **O(1)** | **O(1)** | **O(1)** if duplicates are allowed; **O(n)** otherwise | **O(1)** | **O(1)** if duplicates are allowed; **O(n)** otherwise |
| Position-based **insert()** | Not possible | **O(n)** | **O(1)** if there are no holes; **O(n)** otherwise | Not possible | **O(n)** |
| **remove()** | **O(1)** | **O(1)** | **O(n)** | **O(1)** | **O(n)** |
| Position-based **remove()** | Not possible | **O(n)** | **O(1)** if there are no holes; **O(n)** otherwise | Not possible | **O(n)** |
| **size()** | **O(1)** | **O(1)** | **O(1)** | **O(1)** | **O(1)** |
| **contains()** | **O(1)** | **O(1)** | **O(n)** | **O(1)** | **O(n)** |
| Comparisons (**<=**, **==**, and so on) | **O(n)** | **O(n)** | **O($n^2$)** | **O(n)** | **O($n^2$)** |
| Merges (**|**, **&**, **–**) | **O(n)** | **O(n)** | **O($n^2$)** | **O(n)** | **O($n^2$)** |

There might be holes in an **os_packed_list** if any elements have been removed.

Note that among **O(1)** operations, the constant varies from one structure to another. So, for example, **os_packed_list** insertion with duplicates allowed is fastest, and **os_ixonly** is slowest.

Similarly, the constant for **O(n)** operations varies. So, for example, while position-based insert and remove are **O(n)** for **os_chained_list**, the constant is 1/(the number of pointers per block). So position-based insert and remove are relatively fast operations at small sizes.

Note also that if there are safe cursors open on a particular collection, each insert or remove operation visits each of those cursors and adjusts them if necessary.

## Space Overhead Summary

This section contains a description of the data structures used by each of the following collection representations:

- **os_ordered_ptr_hash** on page 112
- **os_packed_list** on page 114
- **os_ptr_bag** on page 116

These descriptions will allow you to estimate the amount of storage occupied by a collection with a given size and a given representation type. Information on clustering (what is likely to be on the same page as what) is also included to help you predict paging behavior.

### Representation components

Each type of representation has the following components:

- Header
- Entry for each element
- Some number of empty entries

### Representations that support duplicates

Representations that support duplicates also have the following:

- Count slot for each entry
- Some number of empty count slots

The entry for a given element is likely to be on a different page from the collection header. In addition, the count slot for a given element is likely to be stored on a different page from both the header and the entry for the element.

### Estimating the average size of a collection

The number of empty entries and count slots is, on average, proportional to the collection's size. You can estimate the average size of a collection with a given representation using the following pieces of information:

- Average fraction of total entries that are not empty (the *average fill*)
- Size of each kind of component
- Size of the collection

### Formula for calculating size

Here is the formula to use:

```
average total size = size_of(header) + size *
    ( size_of(entry) + size_of(count slot) ) / (average fill)
```

Here is the information you need for each type of representation:

| *Representation Type* | *Header Size in Bytes* | *Entry Size in Bytes* | *Count Slot Size in Bytes* | *Average Fill %* |
|---|---|---|---|---|
| **os_ptr_bag** | **48** | **4** | **4** | **58.3** |
| **os_order_ptr_hash, size <= 65535** | **56** | **8** | **0** | **58.3** |
| **os_order_ptr_hash, size > 65535** | **56** | **12** | **0** | **58.3** |
| **os_packed_list** | **40** | **4** | **0** | **83.3** |

### Minimum representation fills

The minimum fills for each type of representation are as follows:

| *Representation Type* | *Minimum Fill %* |
|---|---|
| **os_ptr_bag** | **46.7** |
| **os_order_ptr_hash, size <= 65535** | **46.7** |
| **os_order_ptr_hash, size > 65535** | **46.7** |
| **os_packed_list** | **66.7** |

### Calculating an upper bound on collection space

You can use this information to get an upper bound on collection space overhead:

```
maximum total size = size_of(header) + size *
    ( size_of(entry) + size_of(count slot) ) / (minimum fill)
```

# Chapter 5
# Queries and Indexes

The information about queries and indexes is organized in the following manner:

# Queries and Indexes Overview

The C++ language makes possible the sort of navigational data access required by typical design applications. But, while navigation provides the most efficient form of access in many circumstances, other situations require associative access. Lookup of an object by name or ID number, for example, is a simple form of associative access. Both associative and navigational retrieval are indispensable to databases supporting complex, data-intensive design applications.

Therefore, among the database services provided by ObjectStore is support for query processing. A query facility with adequate performance must go beyond support for linear searches. So ObjectStore provides a *query optimizer,* which formulates efficient retrieval strategies, minimizing the number of objects examined in response to a query. The query facilities are used from within C++ programs, and they treat persistent and nonpersistent data in an entirely uniform manner.

# Performing Queries with **query()**

To retrieve a collection of those elements that satisfy a specified condition, use the function **os_Collection::query()**.

For more information, see Chapter 2, Collection, Query, and Index Classes, in the *ObjectStore Collections C++ API Reference*.

Declaration

This function is declared

```
os_Collection<E> &query(
    char *type_string,
    char *query_string,
    os_database *schema_database = 0,
    char *file_name = 0,
    os_unsigned_int32 line = 0,
    os_boolean dups = query_dont_preserve_duplicates
) const;
```

where **E** is the element type parameter.

## Example Query

Here is an example:

```
os_database *people_database;
os_Set<person*> *people;

. . .

os_Set<person*> &teenagers = people->query(
        "person*",
        "this->age >= 13 && this->age <= 19",
        people_database
);
```

## Query Arguments

Form of the call

Calls to the function can take the form

```
collection-expression.query(
    element-type-name,
    query-string,
    schema-database
)
```

*collection-expression*

The *collection-expression* in the above example is ***people**, and defines the collection over which the query will be run.

*element-type-name*

The argument *element-type-name*, **person*** in this example, is a string indicating the element type of the collection being queried.

*query-string*

The *query-string* is a C++ *control expression* indicating the query's selection criterion. In this example it is **this->age >= 13 && this->age <= 19**. An element, **e**, satisfies the selection criterion if the control expression evaluates to a nonzero **int** (true) when **e** is bound to **this**.

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

- There are no variables that are not data members.
- There are no function calls, except calls to **strcmp() or strcoll()**, calls involving a comparison operator for which the user has defined a corresponding rank function and / or hash function, and calls to member functions that satisfy the restrictions described in Paths and Member Functions on page 68.

*schema-database*

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

If the transient database is specified, the application's schema (stored in the application schema database) is used to evaluate the query. The application schema database almost always contains the required schema, but it might be closed at the time of the call to **query()**. So using it as the **schema_database** argument might involve the overhead of a database open.

**file_name** and **line** arguments

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

**dups** arguments

If the **dups** argument is the enumerator **query_dont_preserve_duplicates**, duplicate elements that satisfy the query condition are not included in the query result. If **dups** is the enumerator **query_preserve_duplicates**, duplicate elements that satisfy the query condition are included in the query result. Using **query_dont_preserve_duplicates** (the default) typically results in better performance.

Return value

The return value of **query()** refers to a collection that is allocated on the heap. So when you no longer need the resulting collection, you should reclaim its memory with **::operator delete()** to avoid memory leaks. The resulting collection has the same behavior as

the collection being queried. The order of the elements in the result cannot be guaranteed to be the order of the elements in the collection being queried.

## Queries Compared to Collection Traversals

The query above serves as a sort of shorthand for the following collection traversal:

```
os_Set<person*> *people;
. . .

os_Cursor<person*> c(*people);
os_Set<person*> *teenagers =&os_collection::create(
        os_database::get_transient_database()
);

person *p = 0;
for(p = c.first(); c.more() ; p = c.next())
    if (p->age >= 13 && p->age <= 19)
        *teenagers |= p;
os_set *people;
. . .

os_cursor c(*people);
os_set *teenagers =
    &collection::create(os_database::transient_database);
person *p = 0;

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

### Optimizing queries

Queries, however, can be optimized so that, unlike this traversal, they need not involve examination of every element of the collection being queried. This happens when indexes are added to a collection. Query optimization is discussed later in this chapter (see Executing Bound Queries on page 139).

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

```
os_database *people_database;
os_Set<person*> *people;
. . .

os_Set<person*> &teenagers = people->query(
        "person*",
        "age >= 13 && age <= 19",
        people_database
);
```

# Single-Element Queries with **query_pick()**

The sample query above returns a reference to a collection. But some queries are intended to locate just one element. In such cases, it might be more convenient to use **os_Collection::query_pick()**, a query function that returns a single element rather than a collection. Using this form has the additional advantage that more opportunities for optimization are available when it is known that only a single element is sought.

For more information, see Chapter 2, Collection, Query, and Index Classes, in the *ObjectStore Collections C++ API Reference*.

### Declaration

This function is declared

```
E query_pick(char*, char*, os_database*) const;
```

where **E** is the element type parameter.

Calls to **query_pick()** have the same form as calls to **query()**. If using the parameterized version, **os_Collection::query_pick()**, the return value is the the **os_Collection** parameter type. If you are using the nonparameterized version, **os_collection::query_pick()**, the return value is **void***, and you will often have to apply a cast to the result.

## Example query_pick()

Here is an example:

```
os_database * parts_database;
os_Set<part*> *parts;
. . .

part *part_number_411 = parts->query_pick(
        "part*",
        "part_number == 411",
        parts_database
);
```

If more than one element satisfies the query's selection criterion, one of them is picked and returned. So, except for the additional opportunities for optimization, using **query_pick()** is equivalent to calling **os_Collection::pick()** on the result of invoking **query()**.

If no element satisfies the query, **0** is returned.

# Existential Queries with **exists()**

Sometimes a collection is queried to determine whether there exists some element that satisfies the selection criterion, and the identity of the particular element or elements that do satisfy the criterion is not of interest. For such cases, you should use **os_Collection::exists()**. More opportunities for optimization are available when it is known that this is the ultimate intent of the query.

Existential queries are also discussed in Nested Existential Queries on page 134.

Calls to **exists()** have the same form as calls to **query()** and **query_pick()**, but instead of returning a collection or element, they return a nonzero **os_int32** (**int** or **long**, whichever is 32 bits on your platform) for true, and **0** for false.

For more information, see Chapter 2, Collection, Query, and Index Classes, in the *ObjectStore Collections C++ API Reference*.

## Example exists()

Here is an example:

```
os_database *print_request_db;
class page {...int page_number;...};
class print_request {...os_List<page*> *pages;...};
print_request *request;
. . .

if (request->pages->exists(
        "page*",
        "page_number > 100",
        print_request_db
    )
)
request->queue_at_end(print_queue);
```

# Query Functions and Nested Queries

In all three forms of queries, the query string can itself contain queries. A nested collection query has the form

*collection-expression* **[:** *int-expression* **:]**

where *collection-expression* is some element of the top-level integer-expression of type **os_Collection**, and *int-expression* is the selection criterion for the nested query.

A nested single-element query has the form

*collection-expression* **[%** *int-expression* **%]**

where *collection-expression* and *int-expression* are as for nested collection queries.

These nested queries all have the same characteristics as the query expressions discussed in Performing Queries with query() on page 127, except that the collection returned is converted to an **int** by the query processor. The returned collection is converted to **0** (that is, false) if it is empty, and a nonzero **int** (that is, true) if it is nonempty.

For more information, see Chapter 2, Collection, Query, and Index Classes, in the *ObjectStore Collections C++ API Reference*.

## Example Nested Query

Here is a query that finds the musicians among a company's employees:

```
class employee {... os_Set<hobby*>& hobbies; ...};
class hobby {... char *name; ...};
os_Set<employee*> &employees = ...;
. . .

os_Set<employee*> musicians = employees->query(
    "employee*",
    "hobbies[:!strcmp(name, "music"):]",
    db
);
```

In this query, the selection criterion is the query string **hobbies [: !strcmp(name, "music") :]**. Since this is a nested query expression, the collection that it designates is converted to an **int**. The nested query is converted to **0** (false) when it returns an

empty set (there is no hobby named **music**). Otherwise it is converted to a nonzero value.

The query string in the above example is therefore equivalent to

```
[:hobbies[:!strcmp(name, "music"):].size !=0:]
```

# Nested Existential Queries

The collection returned by a nested query is converted to an **int** by the query processor. The returned collection is converted to **0** (that is, false) if it is empty, and a nonzero **int** (that is, true) if it is nonempty.

This is particularly useful for performing existential queries. Consider the following query:

```
os_database *db;
os_Set<part*> &a_set ... ;
a_set->query("part*", "children[:1:]",  db);
```

This query selects all parts in **a_set** that have children. This is because, for each element, **e**, of **a_set**, **e–>children[:1:]** returns **e->children**, which is converted to an integer, and if the integer is nonzero (true), **e** is selected. If **e->children** is empty, it is converted to **0** (false), and so is not selected.

To find the parts in **a_set** that have no children, you can use the following query:

```
os_database *db;
os_Set<part*> &a_set ... ;
a_set->query("part*", "!children[:1:]", db);
```

Here is a query that selects those descendents of **a_part** that have children all of which are primitive. So it selects all descendents that are strictly on the second level from the bottom of the assembly.

```
os_database *db;
part *a_part;

a_part->get_descendents()->query(
    "person*",
    "children[:1:] && !children[:children:]",
    db
);
```

For more information, see Chapter 2, Collection, Query, and Index Classes, in the *ObjectStore Collections C++ API Reference*.

## Example Nested Existential Query

Below is a final example involving nesting of queries. This locates the employee whose child has social security number **123456789**.

```
employees->query_pick(
    "employee*",
    "children[:ss==123456789:]",
    db
);
```

The inner query returns **0** (false), for a given employee, if none of her children has social security number **123456789**. In this case, the employee is not selected. If, for a given employee, at least one of her children does have social security number **123456789**, the inner query returns an **int** greater than **0** (true), and the employee is selected.

# Preanalyzed Queries

It is useful to think of query evaluation as consisting of three logical steps:

1 Analysis of the query expression

2 Binding of the free variable and function references in the query (that is, binding of all identifiers except member names), if any

3 Actual interpretation of the bound query

The first step, analysis of the query expression, is likely to be a relatively expensive operation. If the same query is performed several times, perhaps with different values for the free variables each time, and perhaps on different collections each time, you should use a *preanalyzed query*.

## Creating Query Objects with the os_coll_query Class

To use a *preanalyzed query*, you create a query object, an instance of the class **os_coll_query**. This query will be analyzed upon creation. Subsequently, each time you want to perform the query, you provide bindings for the free variable and function references, and you specify the collection over which the query is to be performed. This way, the cost of analyzing the query is incurred only once for a query that is bound and interpreted several times.

For more information, see Chapter 2, Collection, Query, and Index Classes, in the *ObjectStore Collections C++ API Reference*.

### Form of the call

A preanalyzed query is created with one of the static member functions **os_coll_query::create()**, **os_coll_query::create_pick()**, or **os_coll_query::create_exists()**. Calls to these functions have the form

```
os_coll_query::create(
    element-type-name,
    query-string,
    schema-database
)
```

A **const os_coll_query&** is returned in each case.

**os_coll_query::create()** must be called from within an ObjectStore transaction. An **os_coll_query** object can be created persistently, but it is up to the application to keep track of how to get at it in subsequent transactions (navigable from a root).

## Destroying Query Objects with destroy()

**os_coll_query::destroy** deletes the specified instance of **os_coll_query**. Call **os_coll_query::destroy** only if you are certain a query is no longer needed.

For more information, see Chapter 2, Collection, Query, and Index Classes, in the *ObjectStore Collections C++ API Reference*.

## Function Calls in Query Strings

As with the query strings introduced earlier, the query string here is an **int**-valued expression, and calls to **strcmp()** and **strcoll()** are allowed, as are calls involving comparison operators for which the user has defined a corresponding rank function, and calls to member functions satisfying the restrictions described in Paths and Member Functions on page 68.

For preanalyzed queries, the query string can also include calls to other nonoverloaded global functions, as long as

- The return type of each function is specified explicitly with a cast.
- The function references are bound as described below.
- All function calls involve zero, one, or two arguments, and, for two-argument calls, the first argument is a pointer.

## Creating Bound Queries

Variables (including data members) can appear in a query string, as long as the type of each variable (except data members) is specified explicitly with a cast. Consider, for example:

```
const os_coll_query &age_range_query =
    os_coll_query::create(
        "person*",
        "age >= *(int*)min_age_ptr && age <=*(int*)max_age_ptr",
        db1
    );
```

This creates a preanalyzed query for people in a given age range. Note the type casts used to specify the types of the free variables **min_age_ptr** and **max_age_ptr**.

### Binding a query's variables

Once you have a preanalyzed query, you can create a *bound query* at any time, using the constructor for the class **os_bound_query**. Bound queries must be transiently allocated; they should not be created with persistent **new**.

The bound query constructor takes two arguments: a preanalyzed query, and a *keyword_arg list*, an instance of **os_keyword_arg_list**. Here is an example:

```
int teenage_min_age = 13, teenage_max_age = 19;

os_bound_query teenage_range_query(
    age_range_query, (
        os_keyword_arg("min_age_ptr", &teenage_min_age),
        os_keyword_arg("max_age_ptr", &teenage_max_age)
    )
);
```

This creates a bound query for finding teenagers using the analyzed query in the previous example.

### Comma operator overloading

The comma operator is overloaded in such a way that you can designate a **keyword_arg_list** with an expression of the following form:

```
(
    keyword_arg-expr,
    keyword_arg-expr,
    . . . ,
    keyword_arg-expr
)
```

You create a *keyword_arg* with the constructor for the class **os_keyword_arg**, as in

```
os_keyword_arg("min_age_ptr", &teenage_min_age)
```

This binds the address **teenage_min_age** to the variable **min_age_ptr** in the query string, specifying the value to be used in analyzing the query.

### Binding a query's functions

Just as a query's free variable references must be bound before the query is evaluated, so must all function names. For example, the query:

```
const os_coll_query &the_query = os_coll_query::create(
```

```
    "person*",
    "age >= (int)a_func((int) x)",
    db1
);
```

might be bound with

```
int current_x = 7;

os_bound_query the_bound_query(
    the_query, (
        os_keyword_arg("x", current_x),
        os_keyword_arg("a_func", a_func)
    )
);
```

Note that when a query is evaluated, functions will be invoked an undefined number of times, depending on the evaluation plan formulated by the query optimizer. So, for functions with side effects, the actual results are undefined.

The functions **strcmp()** and **strcoll()** are specially recognized by the query optimizer, so you do not have to bind them.

## Executing Bound Queries

A version of **os_Collection::query()** takes a **const os_bound_query&** argument, as do versions of **os_Collection::query_pick()** and **os_Collection::exists()**.

The bound query can then be used directly in query evaluation, as in

```
people.query(teenage_range_query);
```

Note that, as with the other overloadings of query functions, the return value refers to a collection that is allocated on the heap. So when you no longer need the resulting collection, you should reclaim its memory with **::operator delete()** to avoid memory leaks.

For more information, see Chapter 2, Collection, Query, and Index Classes, in the *ObjectStore Collections C++ API Reference*.

# Indexes and Query Optimization

## Adding an Index to a Collection with add_index()

You can direct ObjectStore to optimize queries over a particular collection by adding an *index* into the collection with the member function **os_collection::add_index()**.

Suppose, for example, you want to optimize lookup of parts in **a_set** by part number, as in the following query:

```
a_set->query_pick("part*", "part_number==411", db1)
```

Example: **add_index()**

You request an index into the set **a_set**. You specify the *key* of the index as the value of the data member **part_number**. To do this, use the member function **os_collection::add_index()**:

```
os_Set<part*> *a_set;
. . .
os_index_path &key_spec =
    os_index_path::create("part*", "part_number",db1);
a_set->add_index(key_spec);
```

The key here is specified with a reference to a path. See Creating Paths on page 65.

The last argument to **add_index()** specifies the database, segment, or object cluster in which the index is stored. The index remains until it is removed with **drop_index()**. By default, an index is placed in the same segment as the collection for which the index is being added.

Once you invoke this function, any query over **a_set** involving lookup by **part_number** is optimized.

Indexes can also end in functions. In this case, the query must end in the function in order to employ the index.

For more information, see Chapter 2, Collection, Query, and Index Classes, in the *ObjectStore Collections C++ API Reference*.

## Index Maintenance

You might need to perform index maintenance for any data member or member function in the path used to specify the index key. See Performing or Enabling Index Maintenance on page 148.

## Pointer-Valued Members and char* Keys

If you create an index with a path to a pointer-valued data member — other than a **char***-valued member — you can optimize lookup based on address. **char***-valued data members are treated specially. An index based on a **char*** member optimizes lookup by the string pointed to.

However, because the query is on the address and not the value, pointer-valued members are limited in their usefulness.

## Indexes and Performance

Without the index, a linear search must be used to perform such queries, and each element of **a_set** will have to be examined. By adding an index into **a_set**, you are instructing the system to maintain an access method (consisting of hash tables and / or a B-tree) allowing efficient lookup by **part_number**.

Adding an index into a collection slows down updates to the collection somewhat. It also slows down updates to the data member specifying the index key. This is because index maintenance is performed whenever such an update occurs. But indexes make the associated lookups significantly faster. So it is a good idea to request an index if the ratio of lookups to updates is large.

## Dropping Indexes from a Collection with drop_index()

Indexes can be added and dropped during the run of an application. If an index makes sense for only part of an application's run, the application can add an index and then remove it later. For example, if the first part of an application performs many lookups but relatively few updates, and the second part performs many updates and relatively few lookups, the program can add an index at the beginning of the first part, and then remove the index at the beginning of the second part.

Because of this, you might unexpectedly find that you want to temporarily drop and later re-add an index. It is a good idea to design your application to keep track of your indexes so you can easily drop and re-add them at a later time, especially if you have many indexes.

Example: **drop_index()**

You remove an index with the member function **drop_index()**. Here is an example:

```
os_Set<part*> *a_set;
. . .
os_index_path &key_spec =
    os_index_path::create("part*","part_number", db1);
. . .
a_set->drop_index(key_spec);
```

Note that you specify the key, with an **os_index_path**, when dropping an index, because the same collection can have several different indexes — to optimize different kinds of lookups.

The **os_index_path** argument does *not* need to be the same instance of **os_index_path** supplied when the index was added, but it must specify the same key. If the path strings used to create two **os_index_path**s differ only with regard to white space, the **os_index_path**s specify the same index key.

If an index with the specified key was never added to the collection, err_no_such_index is signaled.

You can add and drop indexes at run time as frequently as you like. The ObjectStore query optimizer adapts dynamically.

For more information, see Chapter 2, Collection, Query, and Index Classes, in the *ObjectStore Collections C++ API Reference*.

## Testing for the Presence of an Index with has_index()

You can also test for the presence of an index with a specified key, using the member function **has_index()**.

This function returns a value saying whether an index can support the index type specified with **index_options**.

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

Possible values for **index_option** are **ordered** and **unordered**.

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

| | |
|---|---|
| **has_index(path,os_index::ordered)** | Returns true |
| **has_index(path,os_index::unordered)** | Returns true |

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

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

Here is an example:

```
os_Set<part*> *a_set;
. . .
os_index_path &key_spec = ...
. . .
if (a_set->has_index(key_spec options)) ...
```

**options** for **has_index** can have the value **ordered** or **unordered**.

The function **os_collection::get_indexes()** allows you to retrieve information on all the indexes into a specified collection. For more information, see Chapter 2, Collection, Query, and Index Classes, in the *ObjectStore Collections C++ API Reference*.

## Indexes and Complex Paths

Path expressions can specify not just a single data member, but a navigational path involving multiple member accesses. Such path expressions can be used to specify index keys. For example, suppose you want to optimize lookup of a part based on the **emp_id** of any of the **responsible_engineers** for the part (suppose that the member **responsible_engineers** is collection valued). You can use the following path:

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

This path is like ones you have seen before, except that here the data member name **responsible_engineers** is followed by the symbols **[]**, indicating that the next component of the path (**emp_id**) is to be applied to *each element* of the collection of responsible engineers, rather than to the collection itself. Note that you cannot end an expression with the symbols **[]**.

# Index Options

As mentioned above, the index from part numbers to parts is implemented as hash tables (*unordered* indexes). This is the default. But if your application performs *range queries* involving **part_number**, you can request an *ordered* index, implemented using a B-tree. With a B-tree, queries involving **<**, **<=**, **>**, or **>=** comparisons on part numbers can be computed more efficiently than with an index implemented only with hash tables.

Example: B-tree query

Here is an example:

```
part_extent->query(
    "part*", "part_number > 411", db1
)

part_extent[:part_number > 411:]
```

Use of a B-tree also makes iteration in order of **part_number** more efficient (see Controlling Traversal Order on page 73).

## The os_index_path::ordered Enumerator

You request an ordered index by specifying **os_index_path::ordered** as the second argument to **add_index()** when requesting the index:

```
os_Set<part*> *a_set;
. . .
os_index_path &a_path =
    os_index_path::create("part*","part_number", db1);
a_set->add_index(a_path, os_index_path::ordered);
```

Here, **os_index_path::ordered** is an ObjectStore-supplied enumerator.

Of course, if a B-tree is best for parts of your application, and a hash table is best for other parts, you can add and drop indexes of these types dynamically, as appropriate. You cannot have an ordered and unordered index using the same **os_index_path** on a collection.

For more information, see **os_index_path::ordered** in Chapter 2 of the *ObjectStore Collections C++ API Reference*.

## Index Option Enumerators

You can also specify a variety of other index options using various other enumerators. The enumerators can be combined into a bit pattern with bit-wise disjunction (using **|** (bit-wise or)). Here is the complete list of index option enumerators; for additional information see the corresponding entries in Chapter 2, Collection, Query, and Index Classes, in the *ObjectStore Collections C++ API Reference*.

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

duplicate_key is signaled. In addition, for a collection with an index that signals duplicates, inserting an element with the same key value as some other element also provokes an err_index_duplicate_key exception.

- **os_index_path::copy_key**: indicates an index with entries consisting of key-value / element pairs, as opposed to pointer-to-key-value / element pairs (see **os_index_path::point_to_key**, below). For a **copy_key** index, an entry is formed by copying the object at the end of the **os_index_path** that specifies the key. Such an index generally takes up more space than one that points to its keys, but it provides notably faster access times because of reduced paging costs. Specifying both **copy_key** and **point_to_key** for the same index results in a **point_to_key** index.
- **os_index_path::point_to_key**: indicates an index with entries consisting of pointer-to-key-value / element pairs, as opposed to key-value / element pairs. For a **point_to_key** index, an entry includes a pointer to the object at the end of the **os_index_path** that specifies the key. With **point_to_key** behavior, the entire page containing the key is paged in. Because of increased paging costs, such an index generally provides slower access times than an index that copies its keys, but a **point_to_key** index takes up less space. If keys are sparse, for instance, one key contained in a large object, performance can be very slow.

  Specifying both **copy_key** and **point_to_key** for the same index results in a **point_to_key** index.
- **os_index_path::use_references**: indicates a reference-based (as opposed to pointer-based) index. For very large collections, using an **os_ixonly** representation and a reference-based index (or indexes) can, for many operations, significantly reduce address space consumption (see also **os_index_path::copy_key**). Cannot be specified together with **os_index_path::ordered**. Collections using *any* reference-based index must use *only* reference-based indexes.

### Default index behavior

The following disjunction of enumerators specifies the default index behavior:

```
os_index_path::unordered |
os_index_path::allow_duplicates |
os_index_path::copy_key
```

By default, an index is allocated in the same segment as its associated collection. But if you want, you can supply an **os_database*** or **os_segment*** indicating where to allocate a new index. Supply this information as the third argument, for calls that include an options argument. Otherwise, supply the clustering information as the second argument.

# Performing or Enabling Index Maintenance

Whenever you use a path, for each data member mentioned in the path string, except **const** and collection-valued members, you must perform or enable index maintenance. Note that failing to perform or enable index maintenance can result in corrupted indexes, incorrect query results, and program failures.

**os_indexable** data members

Data members declared as **os_indexable** do automatic index maintenance on update. Pointer-valued data members should not be declared as **os_indexable** because the index will be on the address, not the value.

Collections and indexes

For all collections, a collection can either participate in an index, or own an index over itself. In either case, when items are inserted into and removed from collections automatic index maintenance occurs. This is true whether or not the item is **os_indexable** and is also true for indexed member functions. However, you must still do index maintenance on update.

Non-**os_indexable** data members

For data members that are not declared **os_indexable** (for instance pointer-valued data members) you must do index maintenance when you modify the object that the pointer points to.

## Paths as Indexes

If you use a path as an index key, or to specify traversal order for a safe cursor, you have two or three options for each data member in the path.

Option 1: automatic index maintenance

Declare an **os_backptr** member and enable automatic index maintenance. See Declaring an os_backptr Member on page 150. See also Enabling Automatic Index Maintenance on page 152.

This option simplifies the coding of updates, compared to options 2 and 3.

Like option 2, this option carries extra space overhead, compared to option 3, in the form of an **os_backptr** member for each object containing the member.

Option 2: user-controlled index maintenance (using **os_backptr**)

Declare an **os_backptr** and perform user-controlled index maintenance, using **make_link()** and **break_link()**. See Declaring an os_backptr Member on page 150 and User-Controlled Index Maintenance with an os_backptr on page 156.

Like option 3, this option can make coding updates to the member slightly more complex, compared to option 1, but it avoids the use of "wrapper objects" and the apparent value / actual value distinction. See The Actual Value / Apparent Value Distinction on page 154 for more information.

Like option 1, this option carries extra space overhead, compared to option 3, in the form of an **os_backptr** member for each object containing the member.

Option 3: user-controlled index maintenance (without **os_backptr**)

Do not declare an **os_backptr** and perform user-controlled index maintenance, using the collection operations **insert()** and **remove()**. This option only applies if the member is not mentioned in any multistep path used as an index key. See User-Controlled Index Maintenance Without an os_backptr on page 160.

Like option 2, this option can make coding updates to the member slightly more complex, compared to option 1, but it avoids the use of "wrapper objects" and the apparent value / actual value distinction. See The Actual Value / Apparent Value Distinction on page 154 for more information.

With this option, you do not need to declare an **os_backptr,** so you avoid the space overhead, incurred with options 1 and 2, of an **os_backptr** member for each object containing the member. But the index maintenance accompanying each data member update can be more expensive.

Such index maintenance will be more expensive if there is an index that 1) is not keyed by the data member, and 2) indexes a collection on which you perform **insert()** and **remove()** for index maintenance associated with updating the member.

With option 3, each such index will be updated, whereas with options 1 and 2, such indexes are not updated.

# Declaring an **os_backptr** Member

To make a data member indexable, you add to the class whose data member you want to be indexable a public or private data member of type **os_backptr**. The declaration of the data member of type **os_backptr** must *precede* the declaration of the data member (or member functions) you want to make indexable.

Example: **os_backptr** declaration

Here is an example:

```
class part {
    public:
        . . .
        os_backptr b ;
        int id ;
        department *dept ;
        . . .
        part();
        ~part();
        . . .
};
```

Note that it is sufficient to define a single data member of type **os_backptr** for all indexable members of a class.

## Inheritance of the os_backptr

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

Example: **os_backptr** class definitions

Consider, for example, the following class definitions:

```
class base1 {... os_backptr b1 ; ...} ;
class base2a : public base1 {...} ;
class base2b {... os_backptr b2b ;...} ;
class derived : public base2a, public base2b {
    char *name ;
} ;
```

Class **derived**'s name member can use class **base1**'s **os_backptr** member **b1**. Any data member in class **base2a** can also use class **base1**'s **os_backptr** member **b1**. Indexable members in class **base2b** should continue to use **base2b**'s **os_backptr** member **b2b**.

Now consider

```
class base1 {...os_backptr b1 ;...} ;
class base2a : virtual public base1 {...} ;
class base2b {...os_backptr b2b ; ... } ;
class derived : public base2a, public base2b {
    os_backptr d ;
    char *name ;
} ;
```

It is not possible for class **derived**'s indexable data members to use **base1**'s **os_backptr** member, because **base2a** inherits class **base1** virtually. For the same reason, data members in class **base2a** cannot use class **base1**'s **os_backptr** member **b1**. Since **base2b** is not inherited along the leftmost side of the type inheritance lattice, an additional **os_backptr** member (**d** in this example) must be defined to allow **name** to be indexable.

# Enabling Automatic Index Maintenance

Using the functions **os_backptr::break_link()** and **os_backptr::make_link()** whenever you update the member allows ObjectStore to perform index maintenance under user control (see User-Controlled Index Maintenance with an os_backptr on page 156).

This is also true for pointer-valued data members. Because the pointer is the index value, ObjectStore does not detect updates and you must use **os_backptr::break_link()** and **os_backptr::make_link()** for index maintenance whenever you update the member.

But, for all other data members that are not **char*** or **char[]** valued, you can avoid using these functions by declaring the data member using the following macros:

- **os_indexable_member()**
- **os_indexable_body()**
- **os_index()**

If you use these macros, updates to the data member trigger automatic index maintenance.

For more information, see also Chapter 4, Macros and User-Defined Functions, in the *ObjectStore Collections C++ API Reference*.

## The os_indexable_member() Macro

For a data member normally declared as

*value-type member-name* **;**

declare it instead as

**os_indexable_member(***class-name,member-name,value-type***)**
*member-name* **;**

where *class-name* is the name of the class defining the indexable member, *member-name* is the name of the data member being made indexable, and *value-type* is the member's type.

## The os_indexable_body() Macro

The last thing you must do to make a data member indexable is call the macro **os_indexable_body()** to instantiate the bodies of the functions that provide access to the indexable member. These functions ensure that any indexes keyed by that data member are properly updated when the member is. The macro call should appear (at top level) in a file associated with the class defining the indexable member:

```
os_indexable_body(part,id,int,os_index(part,b));
os_indexable_body(part,idx2,int,os_index(part,b));
```

Here, the macro calls have the form

**os_indexable_body(***class***,***member***,***value-type***,***backptr-spec***)**

where

- *class* is the name of the class defining the indexable member.
- *member* is the name of the data member being made indexable and *value-type* is that member's value type.
- *backptr-spec* is a call to the macro **os_index()** indicating the name of the class's **os_backptr** member.

## The os_index() macro

Calls to **os_index()** have the form

**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.

## Avoid White Space in Macro Arguments

Some macro arguments are used (among other things) to concatenate unique names. The details of **cpp** macro preprocessing differ from compiler to compiler, and in some cases it is necessary to enter these macro arguments *without white space* to ensure that the argument concatenation will work correctly. All the examples given in this section follow this important convention, and should therefore work with any **cpp**.

## The Actual Value/Apparent Value Distinction

The actual value of an indexable data member is a special container object that encapsulates the apparent value. For example, the apparent value of the data member **id** of the previous examples is an **int**, but the actual value is an instance of a class defined implicitly by the member macro.

This implicitly defined class defines **operator =()** and a conversion operator (**operator int()** in this example) for converting instances of the implicitly defined type to instances of the apparent value type. Under most circumstances these operators make the container object transparent.

### Examples

For example, to set the **id** of a part to **411**, you simply do

```
a_part->id = 411
```

And to get the value and pass it to a function expecting an **int** argument, you do

```
f(a_part->id)
```

Since **f()** expects an **int** argument (the apparent but not actual value type of **id**), the conversion operator will be invoked, making the above call equivalent to

```
f(a_part->id.operator int())
```

For cases where the actual value cannot be transparent, you should use the functions **getvalue()** and **setvalue()** defined by the actual value type of the indexable member. For example:

```
printf("The id is %d \n", a_part->id);  /* This won't work correctly */
```

will not work because the compiler cannot tell that the second argument to **printf()** is supposed to be an **int**, so the conversion operator is not invoked. Instead you should use

```
printf("The id is %d \n", a_part->id.getvalue());
```

or

```
printf("The id is %d \n", (int)(a_part->id));
```

The **getvalue()** and **setvalue()** functions will always work correctly for getting and setting indexable data member values.

## char* and char() Members

**char*** and **char[]** indexable data members are typically intended to be associated with indexes keyed by string rather than address. For example, iteration based on a **char*** member will proceed in order of the string pointed to rather than in order of address. To ensure proper index maintenance for such members, you must use user-controlled index maintenance. See User-Controlled Index Maintenance with an os_backptr on page 156.

## Restriction on Updates

Note that if the values of an indexable data member are instances of a user-declared class (not pointers to such instances), the values of such an instance's data members cannot be directly altered without circumventing the required index maintenance. To make such a change, the value of the indexable data member must be replaced wholesale with a modified copy of the old value. That is, the instance must be copied and altered, and then the altered object must be copied back as the new value of the indexable member.

# User-Controlled Index Maintenance with an **os_backptr**

Using the macros described in Enabling Automatic Index Maintenance on page 152 allows ObjectStore to perform fully automatic index maintenance for data members. But, for any data member, you can avoid using these macros (and the accompanying actual value / apparent value distinction) by using the functions **os_backptr::break_link()** and **os_backptr::make_link()** whenever you update the member. In this case you need only define the **os_backptr** member; the indexable member itself can be declared in the normal way.

You also use overloadings of **make_link()** and **break_link()** to perform index maintenance for member functions called in query or path strings.

## Making and Breaking Links on Indexable Data Members

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

For indexes keyed by paths that go through the elements of a collection, index maintenance is performed automatically when you change the membership of a collection.

Example: **message** class definition

For example, given the following class definition:

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

class message {
    public:
        . . .
        os_backptr b;
        int id;                    /* an indexable member */
        char *subject_line;    /* a second indexable member */
        class date {
```

```
        public:
            int day;
            int month;
            int year;
        } date_received;    /* a third indexable member */
        . . .
        message(int id, char*subj, int dd, int mm, int yy);
        ~message();
        int set_id(int);
        char *set_subject_line(char*);
        void set_date(int dd, int mm, int yy);
};
```

Example: function definitions

You should define functions for setting each data member as follows:

```
int message::set_id(int i) {
    b.break_link(
        &id,
        &id,
        os_index(message,b) - os_index(message,id)
    );
    id = i;
    b.make_link(
        &id,
        &id,
        os_index(message,b) - os_index(message,id)
    );
    return i;
} /* end set_id() definition */

char *message::set_subject_line(char *subj) {
    b.break_link(
        &subject_line,
        &subject_line,
        os_index(message,b) - os_index(message,subject_line)
    );
    if (strlen(subj) < 500)
        strcpy(subject_line, subj);
    else
        error("string too long");
    b.make_link(
        &subject_line,
        &subject_line,
        os_index(message,b) - os_index(message,subject_line)
    );
    return subj;
} /* end set_subject_line() definition*/

void message::set_date(int dd, int mm, int yy) {
    b.break_link(
        &date_received,
```

```
            &date_received,
            os_index(message,b) - os_index(message,date_received)
        );
        date_received.day = dd;
        date_received.month = mm;
        date_received.year = yy;
        b.make_link(
            &date_received,
            &date_received,
            os_index(message,b) - os_index(message,date_received)
        );
} /* end set_date() definition */
```

Note that since the values of **date_received** are instances of a user-defined class, **date**, this example assumes that you have defined and registered a rank function (and possibly a hash function) for the class **date**. See the examples in Supplying Rank and Hash Functions on page 92.

If these set-value functions provide the only interface for modifying the values of indexable members, indexes will be properly maintained. Circumventing the interface, for example, by passing the address of an indexable member value to a function that alters its value through the pointer, can result in inconsistent indexes.

## Making and Breaking Links to Indexed Member Functions

To maintain indexes keyed by paths containing member function calls, use the following new overloadings of **os_backptr::make_link()** and **os_backptr::break_link()**:

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

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

Automatic index maintenance is not available for such indexes.

### Arguments to **make_link()** and **break_link()**

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

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

**function_name** is the name of the member function itself.

### When to use these functions

Call these functions whenever you perform an update that affects the return value of any member function appearing in a query or path string. You must make a pair of calls (one to **break_link()** and one to **make_link()**) for each such member function affected by each data member change.

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

For indexes keyed by paths that go through the elements of a collection (for example, *** ( (*get_children())[]->get_location() )** ) index maintenance is performed automatically when you change the membership of a collection. See Example: Member Function Calls in Query and Path Strings on page 162.

# User-Controlled Index Maintenance Without an **os_backptr**

Collections do automatic index maintenance. Therefore you avoid **os_backptr** overhead in the following way: If a member is not mentioned in any multistep path used as an index key, you can perform index maintenance by using collection insert and remove operations. Performing index maintenance in this way allows you to avoid declaring an **os_backptr** member. See Performing or Enabling Index Maintenance on page 148.

When you update the data member, follow this procedure:

1 For each index keyed by the member, remove the object containing the member from the indexed collection (it might or might not actually be an element of this collection).

2 Update the member.

3 Insert the object back into each collection, if any, mentioned in step 1, provided the object was a member of that collection prior to your performing step 1.

# Rank and Hash Function Requirements

If your application uses paths ending in instances of a class, you must define and register a rank function and possibly a hash function for the path's terminal type. For ordered indexes keyed by such paths, you must supply a rank function. For unordered indexes keyed by such paths, you must supply both a rank and a hash function (the rank function is used to resolve hashing collisions). See Supplying Rank and Hash Functions on page 92.

# Example: Member Function Calls in Query and Path Strings

Below is a listing of three files that make up a simple program using member function calls in paths and queries:

### Files used in this example

- **rectangle.hh**. This file defines two classes, **coord** and **rectangle**, and includes calls to the **os_query_function()** macro.
- **schema.cc**. This is the schema source file for the program. It contains the calls to **OS_MARK_SCHEMA_TYPE()** as well as to **OS_MARK_QUERY_FUNCTION()**.
- **rectangle.cc**. This file contains the implementation of the member functions of **rectangle**, as well as calls to the **os_query_function_body()** macro. There is also a driver program consisting of the **main()** routine and the function **mquery()**. A rank function for the class **coord** is also provided, to support range queries involving **coord**s.

### The **rectangle** class

The class **rectangle** defines public accessor functions for the following pieces of abstract state:

- Label
- Length
- Width
- Children (a collection of related rectangles)
- Location (an instance of the class **coord**)
- Area

The values for label, length, width, children, and location are stored in private data members. The value for area is computed from the values for length and width.

### Query functions

Each function for reading a piece of abstract state (a *get* function) is declared as a query function. In addition, the public members **coord::x** and **coord::y** are declared as indexable data members. This allows the member function **rectangle::get_location()**, for example, to be called in a query, and allows indexes to be keyed by, for example, the path

```
get_location()->x
```

that is, the *x*-coordinate of a rectangle's location.

## Rectangle Header File — rectangle.hh

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

class coord {
    public:
        os_backptr b ;    /*  needed for indexable member */
        os_indexable_member(coord,x,int) x ;
        os_indexable_member(coord,y,int) y ;
        coord(int x1, int y1) { x = x1 ; y = y1 ; }
        coord() { x = 0 ; y = 0 ; }
} ;

class rectangle {
    private:
        os_backptr b ;     /*  needed for query functions */
        char *label ;
        int length ;
        int width ;
        os_Collection<rectangle*> &children ;
        coord location ;

    public:
        rectangle(
            const char *lbl,
            int l,
            int w,
            const os_Collection<rectangle*> *chldrn_ptr,
            coord lcn
        ) ;

        rectangle(const char *lbl) ;
        ~rectangle() ;
        char *get_label() ;
        void set_label(const char *lbl) ;
        int get_length() ;
        void set_length(int l) ;
        int get_width() ;
        void set_width(int w) ;
        os_Collection<rectangle*> *get_children() ;
        coord *get_location() ;
        void set_location(coord lcn) ;
        int get_area() ;
        static os_typespec *get_os_typespec() ;
} ;

os_query_function(rectangle,get_label,char*) ;
```

```
os_query_function(rectangle,get_length,int) ;
os_query_function(rectangle,get_width,int) ;
os_query_function(rectangle,get_location,coord*) ;
os_query_function(rectangle,get_area,int) ;
os_query_function(rectangle,get_children,\
os_Collection<rectangle*>*) ;
```

Notice that there is no function for setting the children of a given rectangle. This is because the same collection is used to record the children of a rectangle throughout the rectangle's lifetime. Changes in a rectangle's children are reflected by insertions and removals performed on this collection. This means that **rectangle::get_children()** does not need to call **make_link()** and **break_link()**; index maintenance is performed by the collection's insert and remove operations.

## Schema Source File — schema.cc

Here is **schema.cc** containing the calls to **OS_MARK_QUERY_FUNCTION()**.

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

OS_MARK_SCHEMA_TYPE(rectangle);
OS_MARK_SCHEMA_TYPE(coord);
OS_MARK_QUERY_FUNCTION(rectangle,get_label);
OS_MARK_QUERY_FUNCTION(rectangle,get_length);
OS_MARK_QUERY_FUNCTION(rectangle,get_width);
OS_MARK_QUERY_FUNCTION(rectangle,get_location);
OS_MARK_QUERY_FUNCTION(rectangle,get_area);
OS_MARK_QUERY_FUNCTION(rectangle,get_children);
```

## Main Program File — rectangle.cc

### Macro calls and rank functions

Below is the first part of **rectangle.cc**:

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

os_indexable_body(coord,x,int,os_index(coord,b)) ;
os_indexable_body(coord,y,int,os_index(coord,b)) ;

os_query_function_body(rectangle,get_label,char*,b) ;
os_query_function_body(rectangle,get_length,int,b) ;
os_query_function_body(rectangle,get_width,int,b) ;
os_query_function_body(rectangle,get_location,coord*,b) ;
```

```
os_query_function_body(rectangle,get_area,int,b) ;
os_query_function_body(\
rectangle,get_children,os_Collection<rectangle*>*,b) ;

int coord_rank(const void *arg1, const void *arg2) {
    const coord *c1 = (const coord *)arg1 ;
    const coord *c2 = (const coord *)arg2 ;

    if ( c1->x < c2->x )
        return os_collection::LT ;
    else if ( c1->x > c2->x )
        return os_collection::GT ;
    else if ( c1->y < c2->y )
        return os_collection::LT ;
    else if (c1->y > c2->y)
        return os_collection::GT ;
    else
        return os_collection::EQ ;
}
```

Notice the calls to **os_query_function_body()**. The rank function **coord_rank()** is defined here so that we can perform queries comparing **coord** objects, and so we can create an index keyed by a path ending in **coord** objects, in our case the path

```
* ( (*get_children())[]->get_location() )
```

### Rectangle member function Implementations

Here is the second part of **rectangle.cc**:

```
rectangle::rectangle(
    const char *lbl,
    int l,
    int w,
    const os_Collection<rectangle*> *chldrn_ptr,
    coord lcn
):children( os_Collection<rectangle*>::create(
        os_segment::of(this))
) {
    label = new(
        os_segment::of(this),
        os_typespec::get_char(),
        strlen(lbl)+1
    ) char[strlen(lbl)+1] ;

    strcpy(label, lbl) ;
    length = l ;
    width = w ;
    children = *chldrn_ptr ;
    location.x = lcn.x ;
    location.y = lcn.y ;
}

rectangle::rectangle(const char *lbl) :
```

```
    children( os_Collection<rectangle*>::create(
        os_segment::of(this))
) {
    label = new(
    os_segment::of(this),
        os_typespec::get_char(),
        strlen(lbl)+1
    ) char[strlen(lbl)+1] ;
    strcpy(label, lbl) ;
    length = 0 ;
    width = 0 ;
    location.x = 0 ;
    location.y = 0 ;
}

rectangle::~rectangle() {
        delete [] label ;
}

char *rectangle::get_label() {
    return label ;
}

void rectangle::set_label(const char *lbl) {
    b.break_link(this, this, "rectangle", "get_label") ;
    label = new(
        os_segment::of(this),
        os_typespec::get_char()
        strlen(lbl)+1
    ) char[strlen(lbl)+1] ;
    strcpy(label, lbl) ;
    b.make_link(this, this, "rectangle", "get_label") ;
}

int rectangle::get_length() {
    return length ;
}

void rectangle::set_length(int l) {

    /*  two query member functions depend on this data member */
    /*  so we call each of make_link() and break_link() twice */

    b.break_link(this, this, "rectangle", "get_length") ;
    b.break_link(this, this, "rectangle", "get_area") ;
    length = l ;
    b.make_link(this, this, "rectangle", "get_length") ;
    b.make_link(this, this, "rectangle", "get_area") ;
}

int rectangle::get_width() {
    return width ;
}

void rectangle::set_width(int w) {
```

```
        /*  two query member functions depend on this data member */
        /*  so we call each of make_link() and break_link() twice */

        b.break_link(this, this, "rectangle", "get_width") ;
        b.break_link(this, this, "rectangle", "get_area") ;
        width = w ;
        b.make_link(this, this, "rectangle", "get_width") ;
        b.make_link(this, this, "rectangle", "get_area") ;
}

    os_Collection<rectangle*> *rectangle::get_children() {
        return &children ;
    }

    coord *rectangle::get_location() {
        return &location ;
    }

    void rectangle::set_location(coord lcn) {
        b.break_link(this, this, "rectangle", "get_location") ;
        location.x = lcn.x ;
        location.y = lcn.y ;
        b.make_link(this, this, "rectangle", "get_location") ;
    }

    int rectangle::get_area() {
        return length * width ;
    }

    void print_rects(os_Collection<rectangle*> &the_rects) {
        os_Cursor<rectangle*> c(the_rects) ;
        for ( rectangle *r = c.first() ; r ; r = c.next() )
            cout << r->get_label() << "\n" ;
        cout << "\n" ;
    }
```

Note that the set functions perform index maintenance, calling **break_link()** and **make_link()** for each query function whose return value depends on the underlying private data member. For example, **set_width()** calls **break_link()** once for the query function **get_width()** and once for the query function **get_area()**, since both **get_width()** and **get_area()** use the private data member **width** to derive their return values.

As noted earlier, there are no **make_link()** and **break_link()** calls associated with changing a rectangle's children, because index maintenance is performed automatically by the insertion and removal operations of the collection **get_children()** returns.

### The driver

Here is the last part of **rectangle.cc**:

```
    void mquery(os_database *db) {
```

```
os_index_key(coord,coord_rank,0) ;

rectangle *r1 = new(db, rectangle::get_os_typespec())
    rectangle("1") ;
rectangle *r2 = new(db, rectangle::get_os_typespec())
    rectangle("2") ;
rectangle *r3 = new(db, rectangle::get_os_typespec())
    rectangle("3") ;

os_Collection<rectangle*> &the_rectangles =
    os_Collection<rectangle*>::create(db) ;
the_rectangles |= r1 ;
the_rectangles |= r2 ;
the_rectangles |= r3 ;

os_index_path &label_path =
    os_index_path::create("rectangle*", "get_label()", db) ;

os_index_path &length_path =
    os_index_path::create("rectangle*", "get_length()", db) ;

os_index_path &width_path =
    os_index_path::create("rectangle*", "get_width()", db) ;

os_index_path &x_location_path =
    os_index_path::create("rectangle*",
    "get_location()->x",db);

os_index_path &y_location_path =
    os_index_path::create("rectangle*",
    "get_location()->y", db) ;

os_index_path &area_path =
    os_index_path::create("rectangle*", "get_area()", db) ;

os_index_path &children_loc_path =
    os_index_path::create("rectangle*",
    "* ( (*get_children())[]->get_location() )", db) ;

the_rectangles.add_index( label_path,
    os_index_path::unordered) ;

the_rectangles.add_index(length_path,
    os_index_path::ordered) ;

the_rectangles.add_index(width_path,
    os_index_path::ordered) ;

the_rectangles.add_index(x_location_path,
    os_index_path::ordered) ;

the_rectangles.add_index(y_location_path,
    os_index_path::ordered) ;

the_rectangles.add_index(area_path,
    os_index_path::ordered) ;

the_rectangles.add_index(children_loc_path,
```

```
    os_index_path::ordered) ;

r1->set_label("rect1") ;
r1->set_length(20) ;
r1->set_width(60) ;
r1->set_location(coord(10, 20)) ;
r1->get_children()->insert(r2) ;

r2->set_label("rect2") ;
r2->set_length(40) ;
r2->set_width(35) ;
r2->set_location(coord(20, 40)) ;
r2->get_children()->insert(r3) ;

r3->set_label("rect3") ;
r3->set_length(200) ;
r3->set_width(80) ;
r3->set_location(coord(50, 60)) ;

cout << "Rectangles labeled \"rect1\":\n" ;

os_Collection<rectangle*> &answer = the_rectangles.query(
    "rectangle*","strcmp(\"rect1\", get_label()) == 0",db) ;

print_rects(answer) ;
cout << "Rectangles with length < 100:\n" ;

answer = the_rectangles.query("rectangle*",
    "get_length() < 100",db) ;

print_rects(answer) ;
cout << "Rectangles with width > 50:\n" ;

answer = the_rectangles.query("rectangle*",
    "get_width() > 50",db) ;

print_rects(answer) ;
cout << "Rectangles with location x coord <= 25:\n" ;

answer = the_rectangles.query("rectangle*",
    "get_location()->x <= 25",db) ;

print_rects(answer) ;
cout << "Rectangles with area >= 3000:\n" ;

answer = the_rectangles.query("rectangle*",
    "get_area() >= 3000",db) ;

print_rects(answer) ;
cout<<"Rectangles with children whose location < (30, 50):\n";

const os_coll_query &children_loc_query =
    os_coll_query::create("rectangle*",
    "(*get_children())[:*get_location()<*(coord*)a_coord_ptr :]",
    db) ;

coord a_coord(30, 50) ;
```

```
    os_bound_query bound_children_loc_query(
        children_loc_query,
        os_keyword_arg("a_coord_ptr", &a_coord)) ;

    answer = the_rectangles.query(bound_children_loc_query) ;
    print_rects(answer) ;
    os_Collection<rectangle*> *answer_ptr = &answer ;
    delete answer_ptr ;
}
```

### The **main()** driver function

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

    /*  argv[1] is the database */

    cout << "\nStarting mquery ...\n\n";

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

    OS_BEGIN_TXN(tx1, 0, os_transaction::update)
        mquery(os_database::create(argv[1])) ;
    OS_END_TXN(tx1)

    cout << "Done.\n\n" ;
}
```

The function **main()** initializes ObjectStore and ObjectStore collections, and calls **mquery()**, passing in a newly created database. The function **mquery()** registers the **coord** rank function, and then creates three rectangles, adding them to a collection, **the_rectangles**.

Next, **mquery()** creates several index paths involving member functions. Then it adds indexes to **the_rectangles**, specifying the paths as index keys. Then various set functions are used, triggering the index maintenance coded earlier in **rectangle.cc**. The collection of children for two of the rectangles is modified with **rectangle::get_children()** and **os_Collection::insert()**, which triggers index maintenance performed by **insert()**.

Finally, various queries involving member functions are performed on **the_rectangles**.

### Example: driver output

The output looks like this:

```
Starting mquery ...

Rectangles labeled "rect1":
rect1

Rectangles with length < 100:
rect1
rect2
```

```
Rectangles with width > 50:
rect1
rect3

Rectangles with location x coord <= 25:
rect1
rect2

Rectangles with area >= 3000:
rect3

Rectangles with children whose location < (30, 50):
rect1

Done.
```

This driver demonstrates how to express queries with member functions. To verify that index maintenance is being performed properly, you can modify the driver to create more rectangles and insert them into **the_rectangles**. If **the_rectangles** has only three elements, the query optimizer chooses not to use indexes in query evaluation. Here is an example:

```
char s[500];
rectangle *rx = i
rectangle *ry = 0 ;

for (int i = 0 ; i < 200 ; i++) {
    sprintf(s, "%d", i);
    rx = new(db, rectangle::get_os_typespec()) rectangle(s) ;
    the_rectangles |= rx ;
    rx->set_length(i) ;
    rx->set_width(i) ;
    rx->set_location(coord(i, i)) ;
    if (ry)
        rx->get_children()->insert(ry) ;
    ry = rx ;
}
```

# Chapter 6
# Compaction

The information about compaction is organized in the following manner:

# Compaction Overview

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

# Compaction API — **objectstore::compact()**

The programming interface to compaction is provided by a single function, **objectstore::compact().** The function is described in detail in **objectstore::compact()** in Chapter 2 of the *ObjectStore C++ API Reference*.

## Declaration

The function is declared this way:

```
static void objectstore::compact (
    char **dbs_to_be_compacted,
    os_pathname_and_segment_number
            **segments_to_be_compacted = 0,
    char **dbs_referring_to_compacted_ones = 0,
    os_pathname_and_segment_number
            **segs_referring_to_compacted_ones = 0
);
```

## Function arguments

Here, **dbs_to_be_compacted** is a null-terminated array of the pathnames of all the databases that are to be compacted in their entirety. That is, ObjectStore will compact every segment of every database named by an element of **dbs_to_be_compacted**. A simple call to **compact()** can supply only this one argument:

```
char *compact_dbs[NUM_COMPACT_DBS];
. . .
objectstore::compact (compact_dbs);
```

The argument **segments_to_be_compacted**, like **dbs_to_be_compacted**, specifies what data you want compacted, but it does so at segment granularity rather than database granularity. This argument is a null-terminated array of pointers to instances of the class **os_pathname_and_segment_number**. Each such instance identifies a particular segment by encapsulating the pathname of the database containing the segment together with the *segment number* of that segment within the database. The constructor for this class is declared this way:

```
os_pathname_and_segment_number::
    os_pathname_and_segment_number(
            const char *db,
            os_unsigned_int32 seg_number
);
```

You can obtain the segment number of a segment with an **os_segment::get_number** whose value type is a **const char***. So here is

how you might create an **os_pathname_and_segment_number** to identify a particular segment:

```
os_database *db1 = os_database::lookup("/user/parts/db1");
. . .
/* retrieve obj1 from db1*/
. . .
os_segment *seg1 = os_segment::of(obj1);
os_pathname_and_segment_number *seg1_identifier =
    os_pathname_and_segment_number(
            db1,
            seg1->get_number()
);
```

## Cross-Database Pointers and References

The argument **dbs_referring_to_compacted_ones** can be understood as follows. When ObjectStore compacts a segment, it must adjust all pointers and ObjectStore references to the objects in that segment. ObjectStore always adjusts any pointers and references to these objects that are in the same database, but if there are pointers or references to these objects in other databases, you must specify these other databases explicitly. You do this with a null-terminated array of the databases' pathnames.

If you know that the pointers and references to compacted objects are restricted to certain segments in certain databases, you can specify just these segments rather than the entire databases. This can speed up the compaction operation. You do this with the argument **segs_referring_to_compacted_ones**, which is a null-terminated array of pointers to instances of **pathname_and_segment_number**.

## Compaction Example

Here is a program fragment that calls this function with all four arguments (a full program is presented at the end of this section):

```
char *compact_dbs[NUM_COMPACT_DBS];
char *reference_dbs[NUM_REFERENCE_DBS];
os_pathname_and_segment_number *
        compact_segs[NUM_COMPACT_SEGS];
os_pathname_and_segment_number *
        reference_segs[NUM_REFERENCE_SEGS];
```

```
. . .
objectstore::compact(
    compact_dbs, compact_segs, reference_dbs,
        reference_segs
);
```

## Null Termination

Remember that each argument is a *null-terminated* array. In each array, you must set to **0** the element immediately following the last element specifying a database or segment. In addition, it is the caller's responsibility to delete the storage associated with the arguments when the function returns.

The function will signal the exception err_os_compaction if any invalid arguments are supplied.

## Compaction and Transactions

The function **objectstore::compact()** must be invoked outside any ObjectStore transaction. The function itself initiates a transaction, and does all its work within that one transaction. During this time, all the specified databases and segments will be locked, preventing access by other processes.

You can control the amount of time that other applications are locked out of data access by compacting a few segments at a time. For example, to compact a single segment in a particular database, you can use the following code:

```
char* referencing_dbs[2];
os_pathname_and_segment_number* compact_segs[2];
os_pathname_and_segment_number seg(
    "database_foo",
    segment_to_be_compacted->get_number()
);

referencing_dbs[0] = "database_foo" ;
referencing_dbs[1] = 0 ;
compact_segs[0] = &seg ;
compact_segs[1] = 0 ;
objectstore::compact (0, compact_segs, referencing_dbs);
. . .
```

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

## Measuring Unused Space with os_segment::unused_space()

To serve as a rough guide in determining whether a segment needs to be compacted, ObjectStore provides the function **os_segment::unused_space()**:

```
os_unsigned_int32 os_segment::unused_space() const;
```

This function returns the amount of space (in bytes) in the segment not currently occupied by any object. It accounts for space resulting from objects that have been deleted as well as space that cannot be used as a result of internal ObjectStore alignment considerations. Here is an example of its use:

```
if ((float) (seg1->unused_space()) / (float) (seg1->get_size()) > .10)
    compact_segs[i++] = seg1 ;
```

See also **os_segment::unused_space()** in Chapter 2 of the *ObjectStore C++ API Reference*.

## Header File for Compaction

Programs using compaction must include the header file **<ostore/compact.hh>**, and link with the compaction library.

# Compaction Example

Here is a complete program that uses the **compact()** function:

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

extern "C" {
    char* getenv(const char*);
    int strcmp( const char*, const char* );
    int atoi( const char* );
}

static void printUsage() {
    cout << "Usage: apicompact [-s] [-bs] (-d <dbname>)+ \
        (-r <dbname>)+ \n"
        << " (-ds <dbname> <seg>)+ (-rs <dbname> <seg>)+ \n"
        << " -s is for silent operation.\n"
        << " -bs is for batch schema installation.\n"
        << " -d database to compact.\n"
        << " -r database with ref's to compacted data.\n"
        << " -ds database segment to compact.\n"
        << " -rs database segment with ref's to compacted data.\n";
    cout.flush();
}

int apicompact_main(int argc , char* argv[]) {
    char* compact_dbs[16];
    int compact_dbs_no = 0;
    char* reference_dbs[16];
    int reference_dbs_no = 0;
    os_pathname_and_segment_number* compact_segs[16];
    int compact_segs_no = 0;
    os_pathname_and_segment_number* reference_segs[16];
    int reference_segs_no = 0;

    int silent = 0;
    int inc_schema = 1;
    for( int i = 1; i < argc; i++ ) {
        if ( strcmp(argv[i], "-d") == 0 && i < argc - 1 ) {
            i++;
            compact_dbs[compact_dbs_no++] = argv[i];
        }   /* end if */

        else if ( strcmp(argv[i], "-r") == 0 && i < argc - 1 ) {
            i++;
            reference_dbs[reference_dbs_no++] = argv[i];
        }   /* end else *if /

        else if ( strcmp(argv[i], "-ds") == 0 && i < argc - 2 ) {
            compact_segs[compact_segs_no++] =
                new os_pathname_and_segment_number(
```

```
                argv[i+1], atoi( argv[i+2]));
            i += 2;
        }    /* end else if */

        else if ( strcmp(argv[i], "-rs") == 0 && i < argc - 2 ) {
            reference_segs[reference_segs_no++] =
                new os_pathname_and_segment_number(
                argv[i+1], atoi( argv[i+2]));
            i += 2;
        }   /* end else if */

        else if ( strcmp( argv[i], "-s" ) == 0 ) {
            silent = 1;
        }    /* end else if */

        else if ( strcmp( argv[i], "-bs" ) == 0 ) {
            inc_schema = 0;
        }    /* end else if */

        else {
            printUsage();
            cout << "No option \"" << argv[i] << "\"\n";
        } /* end else */

    }    /* end for loop */

    compact_dbs[compact_dbs_no] = 0;
    reference_dbs[reference_dbs_no] = 0;
    compact_segs[compact_segs_no] = 0;
    reference_segs[reference_segs_no] = 0;

    if( !silent )
        cout << "Starting " << argv[0] << endl << flush;

    objectstore::initialize();

    objectstore::set_incremental_schema_installation(
        inc_schema );

    objectstore::compact (
        compact_dbs, compact_segs,
        reference_dbs, reference_segs);

  if( !silent )
      cout << "Finished " << argv[0] << endl << flush;
  return 0;

} /* end apicompact_main */
```

# Compactor Limitations

The compactor operates under the following restrictions.

The compactor compacts all C and C++ persistent data, including ObjectStore collections, indexes, and bound queries, and correctly relocates pointers and all forms of ObjectStore references to compacted data.

ObjectStore **os_reference_local** references are relocated assuming that they are relative to the database containing them.

The compactor respects ObjectStore clusters, in that compaction ensures that objects allocated in a particular cluster remain in the cluster, although the cluster itself can move as a result of compaction.

## Restrictions on Compaction Use

The following data restrictions must be observed in using this compactor:

- Union discriminant functions require access to the representation to be compacted in order to run.
- The classic example of a data structure that might require user transformation is a hash table that hashes on the offset of an object within a segment. Since compaction modifies these offsets, there is no way such an implicit dependence on the segment offset can be accounted for by compaction. Of course, transformation of ObjectStore collections is supported in the compactor. Invocation of user data transforms is not currently supported.
- Since the ObjectStore **objectstore::retain_persistent_addresses()** facility requires that persistent object locations within a segment remain invariant, no client application using this facility and referencing segments to be compacted can run concurrently with the ObjectStore compactor.
- Transient ObjectStore references into a compacted segment become invalid after compaction completes.

# File Systems and Compaction

ObjectStore supports two file systems for storing databases, and the compactor can run against segments in databases in either file system.

## File Databases

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

## Rawfs Databases

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

# Compaction Utility

In addition to the programming interface described in Compaction API — objectstore::compact() on page 175, ObjectStore provides an executable, **oscompact**, that can be used to compact specified databases and segments. See oscompact: Compacting Databases in Chapter 4 of *ObjectStore Management* for information on the **oscompact** utility.

# Chapter 7
# 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 allow 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 like **int**) are instances of **os_class_type**, which is derived from **os_type**. (See the hierarchy diagram in The Metatype Hierarchy on page 204).

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 metaobject protocol (MOP) but are not metatypes and are not, therefore, part of the metatype hierarchy. Descriptions of these auxiliary classes begin on page 206.

Besides telling you how to perform run-time read access to ObjectStore application, compilation, and database schemas, this chapter explains how 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 collection is being used), followed by **<ostore/mop.hh>**.

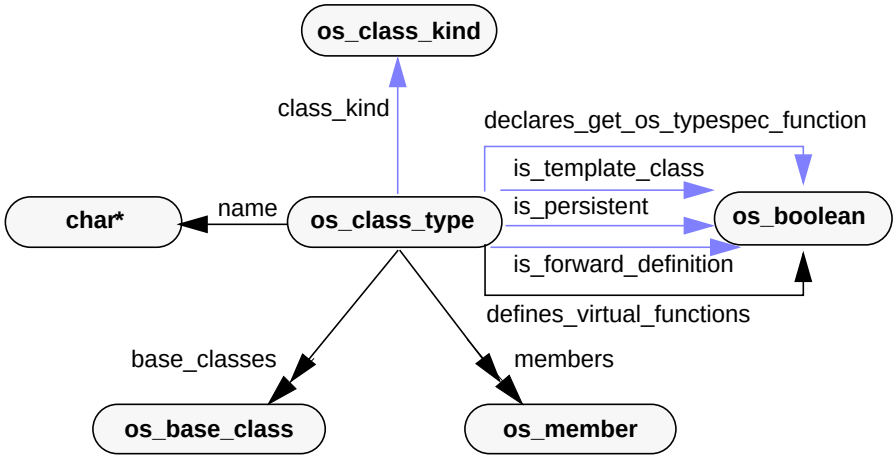
# Attributes of MOP Classes

It is useful to think of classes in the metaobject protocol as having *attributes*, that is, pieces of abstract state that you can access 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. And you update an attribute with a set function. Most of the functions in the metaobject protocol are create, get, or set functions, members of the class whose state they access.

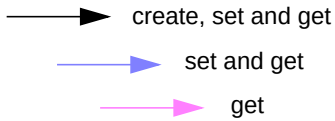
## Attributes of **os_class_type**

Consider for example the class **os_class_type**, whose instances represent C++ classes. Here 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_member**s 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 what type of access to the corresponding attribute 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. And 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*()

and

**set**_*attribute-name*()

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

*attribute-name*()

and

**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. Since set functions only take non-**const this** arguments, direct updates to application, compilation, and database schemas are prevented (assuming 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 199). 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 Schema Installation and Evolution Using MOP on page 202).

### 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 199)

In either case, a reference or pointer to a non-**const** object is returned. The get functions, when performed on non-**const** objects, will 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. That way you can, for example, create an **os_member** without at 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. But note that a nonnull value for the attribute might have to be supplied before installation or evolution, in order to meet the consistency requirements.

For an attribute that must have a nonnull value in order 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 The is_unspecified() function on page 209).

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

## The 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:

**type_at()** 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 co-located 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 would return an **os_type** representing this other object.

Example: **type_at()**

Here is an example of the function's use:

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

## The type_containing() Function

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

**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**

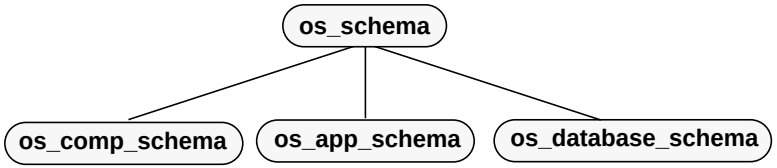
might point to a subobject or data-member value embedded in the middle of the object whose type is returned.

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 **1**.

# 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&);
static const os_comp_schema &os_comp_schema::get(
    const database&);
static const os_database_schema &os_database_schema::get(
    const database&);
```

### 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*) const;
```

### Retrieving class types

You can retrieve the **os_class_type**s in a given *compilation* schema this way (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_type**s in a given *application* schema this 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 this 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 this 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 the *ObjectStore C++ API Reference*.

# The Transient Schema

The metaobject protocol provides the ability to 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. So before calling this function, you must perform these steps:

Preparation for the copy operation

1 First, 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 Next, before calling **copy_classes()**, you must create a set to hold the pointers to the classes you want to copy, as in

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

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

- 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") ;
```

- The function **find_type()** can return **0**, so check the result, as in

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

- Then dereference the result of **find_type()** and convert it from a **const os_type&** to a **const os_class_type&**, as in

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

- Finally, insert the class into the set, as in

```
to_be_copied_to_transient_schema |=
    the_const_class_os_collection ;
```

4 Once this is done 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 have copied 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) ;
```

Notice 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 it.

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 The Class os_member_variable on page 229):

```
assert (the_non_const_type_os_collection) ;

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

# Schema Installation and Evolution Using MOP

**os_database_ schema::install()**

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) ;
```

**os_mop::get_ transient_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() ;
```

**os_database_ schema::get_for_ update()**

The **this** argument is a pointer to the schema you want to modify. Notice that **install()** function cannot take a **const this** argument. So 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&)
```

Example: schema install

So a call to **install** might look like

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

Nondefault behavior

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 *ObjectStore C++ API Reference* for a description of the specific installation options available.

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

**os_schema_ evolution::evolve()**

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

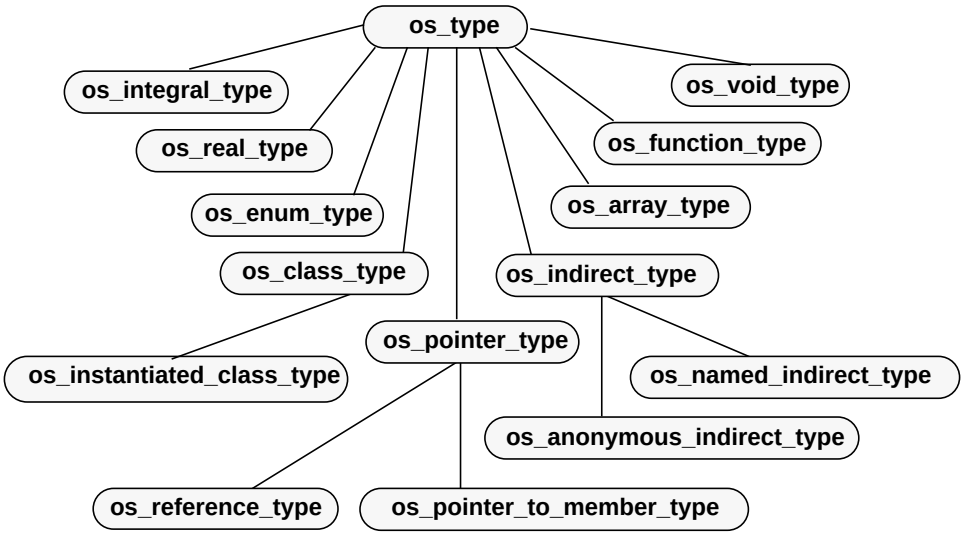
```
static void evolve(
    const char *workdb_name,
    const char*database_name,
    os_schema &new_schema
) ;
```

The **new_schema** argument should be the transient schema. So 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**, also derived from **os_pointer_type**.

Types with **const** or **volatile** specifiers are represented by instances of **os_anonymous_indirect_type**, and **typedef**s are represented by instances of **os_named_indirect_type**.

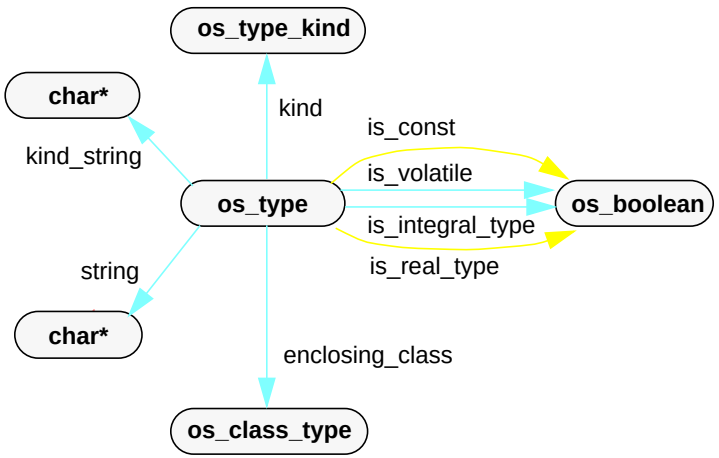
All the classes in the metatype hierarchy are documented in Chapter 2, Class Library, of the *ObjectStore C++ API Reference*. Note that these are not all the types in the metaobject protocol. There are several classes in the metaobject protocol whose instances represent schema objects other than types, such as **os_base_class** (see The Class os_base_class on page 223), and **os_member** and its subtypes (see The Class os_member on page 226). The rest of this chapter contains a section on each class in the protocol.

# The Class **os_type**

Below is a diagram showing the attributes of the class **os_type**. Each arrow represents an attribute and points to its value type. 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 *ObjectStore C++ API Reference* for a complete description of this class.

Attributes of **os_type**




## Create Functions

This class defines no create functions. You always create an instance of **os_type** using a create function defined by one of the subtypes of **os_type**.

## The 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 using the type-safe conversion operators. See Type-Safe Conversion Operators on page 209.

## 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 containing an expression designating the specified type (like **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

The **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 ;
```

The **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 ;
```

The **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 ;
```

The **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 ;
```

The **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 functions, **get_enclosing_class()**, 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.

The **strip_indirect_types()** function

There are also functions, **os_type::strip_indirect_types()**, declared

```
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()** will return an **os_type** representing the type **int**. If the

specified **os_type** represents the type **char const * const**, **strip_indirect_types()** will return an **os_type** representing the type **char const ***.

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

This function calls itself recursively, until the result is not an **os_indirect_type**. So, for example, consider an **os_named_indirect_type** representing

```
typedef const part const_part
```

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

### The **is_unspecified()** function

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

In the transient schema, if such an attribute lacks a value (because you have not yet specified it), the get function returns the unspecified type. This is the only **os_type** for which 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&** — provided 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 ;
```

# The Class **os_integral_type**

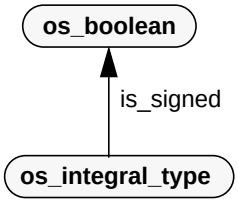
Instances of this class represent one of the following types:

**int**
**unsigned int**
**short**
**unsigned short**
**long**
**unsigned long**
**char**
**signed char**
**unsigned char**

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

Attribute of **os_integral_type**

This class defines one attribute, **is_signed**:




## Create Functions

This 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 signed) ;

static os_integral_type &create_short(os_boolean signed) ;

static os_integral_type &create_int(os_boolean signed) ;

static os_integral_type &create_long(os_boolean signed) ;
```

## Determining a Signed Type with is_signed()

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

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

# The 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 *ObjectStore C++ API Reference* for a complete description of this class.

## Create Functions

This class has three create functions, one for each of the real types listed above.

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

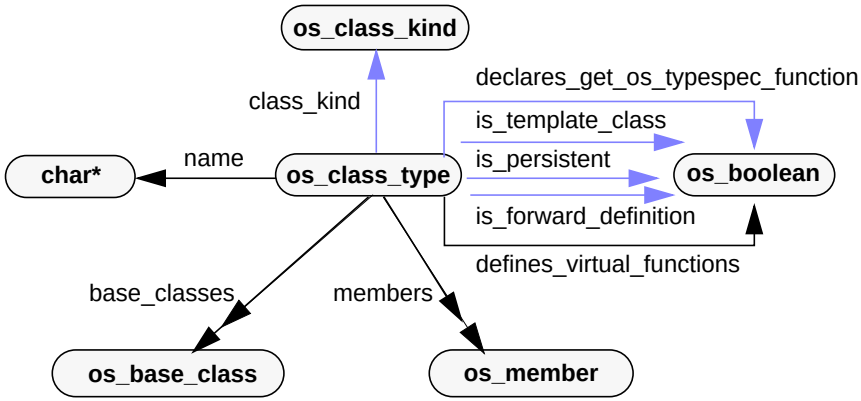
# The Class **os_class_type**

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 *ObjectStore C++ API Reference* for a complete description of this class.

### Attributes of **os_class_type**

Below is a diagram showing 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

This class has two create functions:

```
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** |

| *Attribute* | *Value* |
|---|---|
| **class_kind** | **os_class_type::Class** |
| **defines_get_os_typespec_function** | **0** |
| **is_template_class** | **0** |
| **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** |

## The 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

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

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

## The 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 are:

**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) ;
```

## The members Attribute

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

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

Each instance of **os_member** represents a member of the class (see The Class os_member on page 226). 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 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 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 with a create function; see Create Functions on page 213.

## 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 The Class os_base_class on page 223). 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 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 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 with a create function; see Create Functions on page 213.

## The declares_get_os_typespec_function Function

You can determine if 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.

## The 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 will declare such a member function; if you supply **0**, it will not.

## The 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 zero 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, or installation will fail.

### 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 can also initialize **defines_virtual_functions** with a create function; see Create Functions on page 213.

## The 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) ;
```

## The 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) ;
```

## The is_persistent Attribute

In order to be installed into 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 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 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 Name of this with find_member()

The **find_member()** function returns the member value of **this** that has the specified name:

```
const os_member_variable
      *find_member(const char *name) const ;

os_member_variable *find_member(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, as well as 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, described below.

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

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

```
<err-0008-0010>Unable to get the most derived class in
os_class_type::get_most_derived_class() containing the
                        address 0x[[some-address]].
```

### Example: **get_most_derived_class()**

Here 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** has the results shown in the following table:

| object | most_derived_object | os_class_type |
|---|---|---|
| pic | pC | C |

| object | most_derived_object | os_class_type |
|---|---|---|
| pid | pD | D |
| pib | pD | D |

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

### Example: class hierarchy

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

Given

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

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

Function result

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

If **B1** is instead defined as

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

and

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

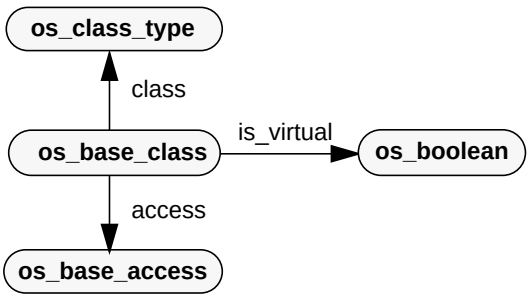
**pb0** and **pi** have the same value because of the optimization, but the base class, **B0**, is collocated with an **int**-valued data member rather than a class-valued data member. **get_most_derived_class()** called on **pb0** or **pi** returns a pointer to the **os_class_type** for the class **C1**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

# The 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 *ObjectStore C++ API Reference* for a complete description of this class.

Attributes of **os_base_class**




## Create Functions

This class has one create function:

```
static os_base_class &create (
    os_unsigned_int32 access ,
    os_boolean is_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 ) ;
```

## The 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 The is_unspecified() function on page 209).

### Setting the attribute

You can set the **class** of an **os_base_class** with

```
void set_class(os_class_type&) ;
```

### Initialization

You can initialize the **class** attribute with **os_base_class::create()**.

## The 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_int_32) ;
```

Initialization — You can initialize the **access** attribute with **os_base_class::create()**.

## The 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. The following function returns the value of the **is_virtual** attribute:

Getting the attribute

```
os_boolean is_virtual() const ;
```

Setting the attribute — You can set this attribute with

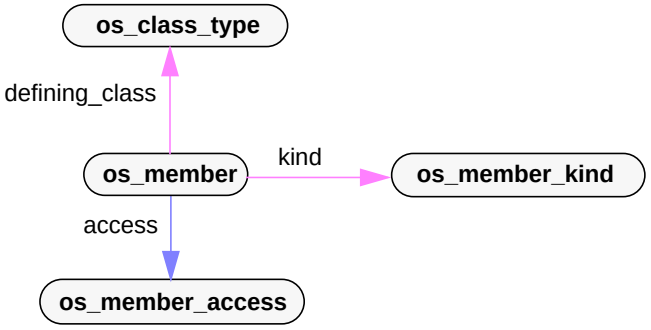
```
void set_is_virtual(os_boolean) ;
```

# The Class **os_member**

An instance of **os_member** represents a data member or member function.

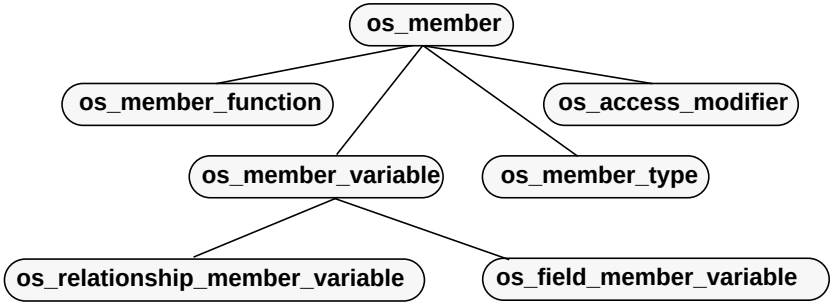
See **os_member** in Chapter 2 of the *ObjectStore C++ API Reference* for a complete description of this class.

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

Since this class 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**.

## The 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) ;
```

If the specified **os_unsigned_int32** does not have the same value as one of the above enumerators, err_mop is signaled.

Initialization

This attribute is initialized to **os_member::Public** by the create functions for the subclasses of **os_member**.

## The kind Attribute

Attribute enumerators

The **kind** of an **os_member** is one of the following enumerators:

| | |
|---|---|
| **os_member::Variable** | For data members |
| **os_member::Function** | For member functions |
| **os_member::Type** | For nested classes |
| **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 ;
```

## The 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 The is_unspecified() function on page 209).

### Setting the attribute

The **defining_class** of an **os_member** is automatically set 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_member**s 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_member**s 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&() ;
```

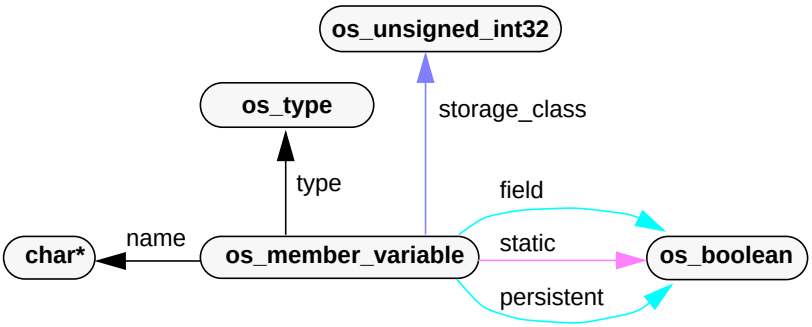
If an attempt is made to perform an inappropriate conversion, err_mop_illegal_cast is signaled.

# The Class **os_member_variable**

Attributes of **os_member_variable**

This diagram shows some of the important attributes of **os_member_variable.**

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




## Create Function

This class has one create function:

```
static os_member_variable &create (
    const char *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** |

## The 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()**.

## The type Attribute

The **type** of a given **os_member_variable** is its value type. You can retrieve an **os_member_variable**'s type with

Getting the attribute

```
const os_type &get_type() const ;
os_type &get_type() ;
```

Setting the attribute — You can set the type with

```
void set_type(os_type&) ;
```

Initialization — You can initialize **type** with **os_member_variable::create()**.

## The 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) ;
```

## The 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 ;
```

## The 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 ;
```

## The 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_variable**s 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_variable**s to non-**const** references to an instance of a subtype.

```
operator os_field_member_variable&() ;
operator os_relationship_member_variable&() ;
```

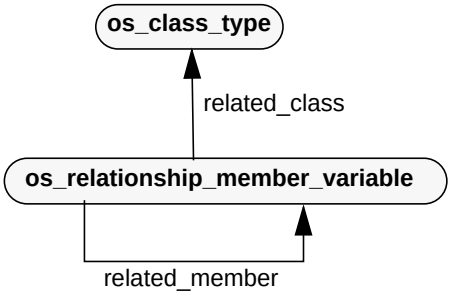
These functions signal err_mop_illegal_cast if an attempt is made to perform an inappropriate conversion.

# The Class **os_relationship_member_variable**

Attributes of **os_relationship_member_variable**

Here is a diagram showing some of the important members of **os_relationship_member_variable**. Note that attributes inherited from **os_member_variable** and **os_member** are not shown.

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




## Create Function

This class has one create function:

```
static os_relationship_member_variable &create (
    const char *name ,
    const os_type *value_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()**.

## The 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&) ;
```

## The 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

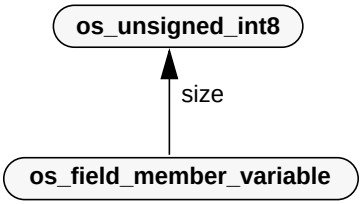
You can set the **related_member** with

```
void set_related_member(os_relationship_member_variable&) ;
```

# The Class **os_field_member_variable**

Attribute of **os_field_member_variable**

Here is a diagram of the attributes of **os_field_member_variable**. Note that attributes inherited from **os_member_variable** and **os_member** are not shown.




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

## Create Functions

This class has one 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()**.

## The 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

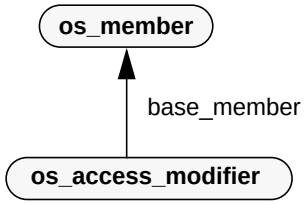
You can set the **size** with

```
void set_size(os_unsigned_int8) ;
```

# The Class **os_access_modifier**

Attribute of **os_access_modifier**

An **os_access_modifier** represents an access modification performed by a class on an inherited member. Note that attributes inherited from **os_member** are not shown.




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

## Create Function

This class has one create function:

```
static os_access_modifier &create(os_member*) ;
```

Function argument

The argument is used to initialize the **base_member** attribute.

## The 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 this with

```
const os_member &get_base_member() const ;
os_member &get_base_member() ;
```

Setting the attribute

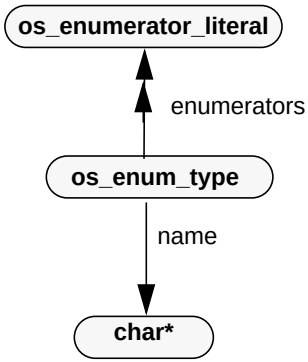
You can set this with

```
void set_base_member(os_member&) ;
```

# The Class **os_enum_type**

Attributes of **os_enum_type**

Here is a diagram showing the attributes of **os_enum_type**. Note that the attributes inherited from **os_type** are not shown.




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

## Create Function

This class has one 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.

## The 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) ;
```

## The 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

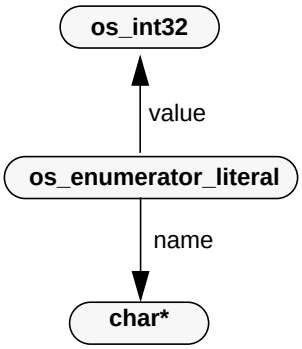
You can set the value with

```
void set_enumerators(const os_List<os_enumerator_literal*>&) ;
```

# The Class **os_enumerator_literal**

Attributes of **os_enumerator_literal**

Instances of this class represent enumerators. Here is a diagram showing the attributes of **os_enumerator_literal**:




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

## Create Function

This class has one create function:

```
static os_enumerator_literal &create (
    const char *name,
    os_int32 value
) ;
```

Function arguments

The arguments specify the initial values for **name** and **value**.

## The 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 the **name** with

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

Setting the attribute

You can set the **name** with

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

Initialization

The **name** is initialized by the create function.

# The 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 *ObjectStore C++ API Reference* for a complete description of this class.

## Create Function

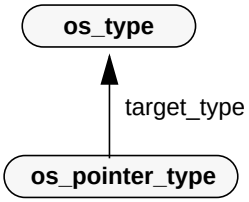
This class's create function is

```
static os_void_type &create() ;
```

# The Class **os_pointer_type**

Attribute of **os_pointer_type**

Instances of this class are used to represent pointer types.




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

## Create Function

The create function for this class is

```
static os_pointer_type &create(os_type *target_type) ;
```

Function argument

The argument is used to initialize the attribute **target_type**.

## The 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 The is_unspecified() function on page 209).

Setting the attribute

You can set the **target_type** with

```
void set_target_type(os_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_type**s 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_type**s to non-**const** references to an instance of a subtype.

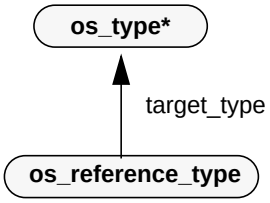
```
operator os_pointer_to_member_type&() const ;
operator os_reference_type&() ;
```

These functions signal err_mop_illegal_cast if an attempt is made to perform an inappropriate conversion.

# The Class **os_reference_type**

Attribute of **os_reference_type**

Instances of this class are used to represent reference types. Note that the following diagram shows attributes inherited from **os_pointer_type**, but not those inherited from **os_type**.




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

## Create Function

The create function for this class is

```
static os_reference_type &create(os_type *target_type) ;
```

Function argument

The argument is used to initialize the attribute **target_type**.

## The 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**.

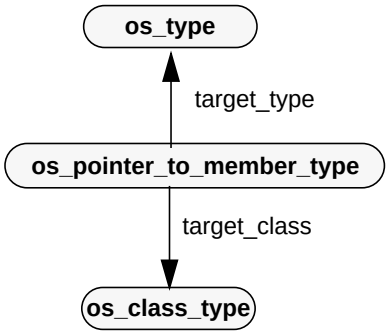
Getting and setting this attribute

You can get and set the **target_type** with functions inherited from **os_pointer_type**.

# The Class **os_pointer_to_member_type**

Attributes of **os_pointer_to_member_type**

Instances of this class are used to represent pointer-to-member types. Note that the following 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 *ObjectStore C++ API Reference* for a complete description of this class.

## Create Function

The create function for this class is

```
static os_pointer_to_member_type &create (
    os_type *target_type,
    os_class_type *target_class
) ;
```

Function argument

The argument is used to initialize **target_class** and **target_type**.

## The 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**.

## The target_class Attribute

The value of this attribute for a given **os_pointer_to_member_type** is the class defining the pointed-to member.

### 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 The is_unspecified() function on page 209).

### Setting this attribute

You can set the **target_class** with

```
void set_target_class(os_class_type&)
```

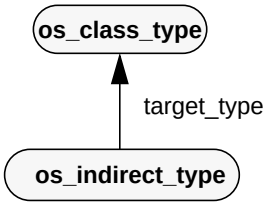
### Initialization

This attribute can be initialized with the create function as well.

# The Class **os_indirect_type**

Attribute of **os_indirect_type**

Instances of this type are direct instances of either **os_indirect_type** or **os_anonymous_indirect_type**.



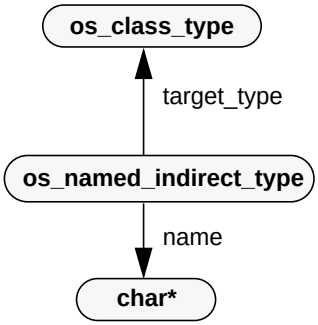

See **os_named_indirect_type** (page 246) and **os_anonymous_indirect_type** (page 248).

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

# The Class **os_named_indirect_type**

Attribute of **os_named_indirect_type**

An instance of this class represents a C++ **typedef**. The diagram below 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 *ObjectStore C++ API Reference* for a complete description of this class.

## Create Function

The create function for this class 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**.

## The 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&)
```

Initialization — This attribute can be initialized with the create function.

## The 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

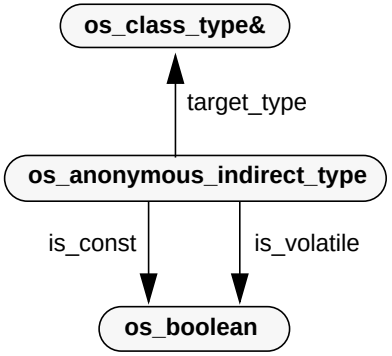
```
void set_name(const char *name) ;
```

Initialization — **name** can be initialized by the create function as well.

# The Class **os_anonymous_indirect_type**

Member functions of **os_anonymous_indirect_type**

An instance of this class represents a **const** or **volatile** type. The diagram below 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 *ObjectStore C++ API Reference* for a complete description of this class.

## Create Function

The create function for this class is

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

Function argument

The argument is used to initialize **target_type**.

## The 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.

## The 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) ;
```

## The 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 ;
```

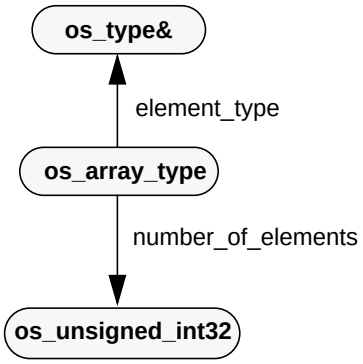
Setting the attribute — You can set it with

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

# The Class **os_array_type**

Attributes of **os_array_type**

Instances of this class are used to represent array types. Note that attributes inherited from **os_type** are not shown.




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

## Create Function

This class has one 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**.

## The 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) ;
```

## The 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&) ;
```

# 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 will end up accessing the wrong address.

## The os_fetch() Functions

Note that the **os_fetch()** functions store a reference to the fetched value in the argument **value**, and also return the value.

The first parameter to **os_fetch** and **os_store** 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 will end up accessing the wrong address.

For more information, see **::os_fetch()** in Chapter 3 of the *ObjectStore C++ API Reference*.

```
void* os_fetch(
    const void *p, const os_member_variable&, void *&value);

unsigned long os_fetch(
    const void *p, const os_member_variable&,
    unsigned long &value);

long os_fetch(
    const void *p, const os_member_variable&, long &value);

unsigned int os_fetch(
    const void *p, const os_member_variable&,
    unsigned int &value);

int os_fetch(
    const void *p, const os_member_variable&, int &value);
```

```
unsigned short os_fetch(
    const void *p, const os_member_variable&,
    unsigned short &value);

short os_fetch(
    const void *p, const os_member_variable&, short &value);

unsigned char os_fetch(
    const void *p, const os_member_variable&,
    unsigned char &value);

char os_fetch(
    const void *p, const os_member_variable&, char &value);

float os_fetch(
    const void *p, const os_member_variable&, float &value);

double os_fetch(
    const void *p, const os_member_variable&, double &value);

long double os_fetch(
    const void *p, const os_member_variable&, long double &value);
```

## The os_store() Functions

For more information, see **::os_store()** in Chapter 3 of the *ObjectStore C++ API Reference*.

```
void os_store(
    void *p, const os_member_variable&, const void *value);

void os_store(
    void *p, const os_member_variable&,
    const unsigned long value);

void os_store(
    void *p, const os_member_variable&, const long value);

void os_store(
    void *p, const os_member_variable&, const unsigned int value);

void os_store(
    void *p, const os_member_variable&, const int value);

void os_store(
    void *p, const os_member_variable&,
    const unsigned short value);

void os_store(
    void *p, const os_member_variable&, const short value);

void os_store(
    void *p, const os_member_variable&,
    const unsigned char value);

void os_store(
    void *p, const os_member_variable&, const char value);
```

```
void os_store(
    void *p, const os_member_variable&, const float value);

void os_store(
    void *p, const os_member_variable&, const double value);

void os_store(
    void *p, const os_member_variable&, 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 metaobject protocol. 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 metaobject protocol — schema information must be accessed in order to identify, *at run time,* the members of the specified object, so their values can be fetched and presented. Something like this functionality might be offered by a browser or debugger.

## The Top-Level print() Function

This example involves several functions. The function **print()** below 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 below).

**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 */
```

Let us explain the function by considering some sample input.

### Sample input: class definitions

Suppose an application uses the classes **part**, **mechanical_part**, and **date**, 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 like this:

```
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()**:

```
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**). Since this is **0**, you have just started printing an object, and so 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 this:

```
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 using **os_base_class::get_offset()**.

**How the object's type is obtained**

The type is obtained 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, since 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 will be 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. Since 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. Since **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()
```

Since you are within the recursive execution, the specified class is **part**. You test the kind of each member 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 below).

**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&**, as well as to all the other subtypes of **os_member**.

You get the value type of the member using **os_member_variable::get_type()** and you get the name of this type 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 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);
```

Here is how this function is defined:

```
/* 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

This function begins by retrieving the value type of the specified data member, and applying **strip_indirect_types()**. The result will be 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 would be

```
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 will put us 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:

```
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.

## The print_a_pointer() function

This function determines that the value type of the current member is a pointer type, and 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 below). 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()**

So the sample output might end up like this:

```
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.

### The **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() */
```

The **print()** function for an array

Print 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 */

    }
```

The **print()** function for a pointer

Print 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 ;

}
```

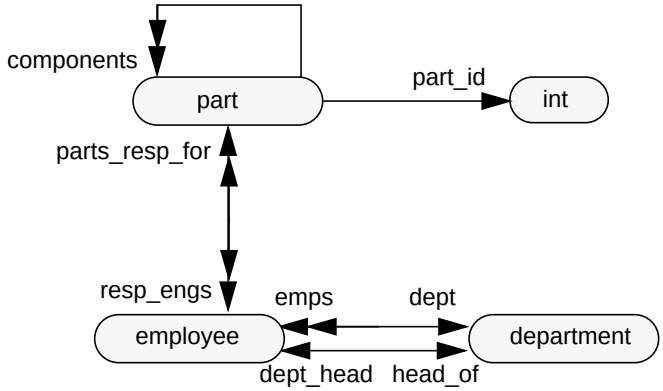
# Example: Dynamic Type Creation

Here is an example of using the metaobject protocol 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 would be 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 would translate 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. Here is what the arrows mean:

- 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 ;
} ;
```

Note that 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 arrays of classes are not used). Note also that 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&**.

## The 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

Here 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

Here 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 that we have defined:

- **ensure_in_trans()** (defined on page 275)
- **copy_to_trans()** (defined on page 274)
- **add_single_valued_member()** (defined on page 275)
- **add_many_valued_member()** (defined on page 276)
- **add_member()** (defined on page 273)

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. **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. Note, however, that built-in C++ types, like **int**, are already present in the transient schema.

Notice also that lookups in the application schema and copying from the application schema must be performed within a transaction, since 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:

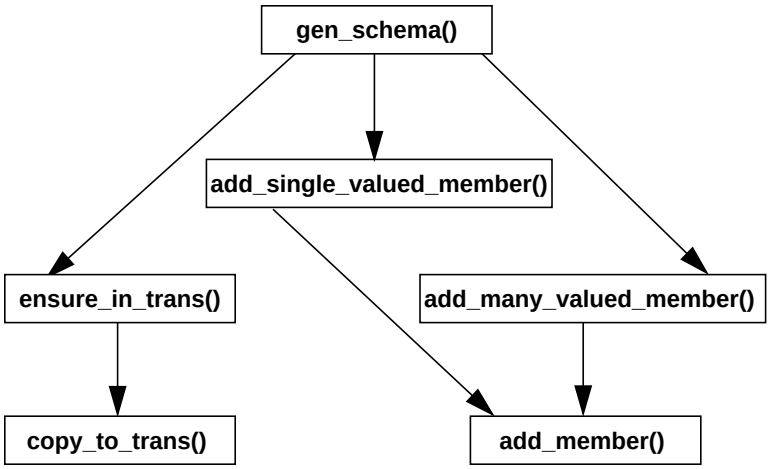
- **end_1** has a single arrow and **end_2** has no arrow.
- **end_1** has no arrow and **end_2** has a single arrow.
- **end_1** has a double arrow and **end_2** has no arrow.
- **end_1** has no arrow and **end_2** has a double arrow.
- **end_1** has a single arrow and **end_2** has a single arrow.
- **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 where an **arc** has arrows at both ends. For now, the example just creates regular data members.

Each data member is created as well as 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()**. **add_member()** adds a specified member to a specified class.

## Call Graph of Non-ObjectStore Functions for gen_schema()




Once **gen_schema()** finishes processing all the **arc**s, the transient schema contains the schema represented by the diagram.

## The gen_schema.cc Source File

Here 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.h>
#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) ;

    }
```

Notice that **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) ;
}
```

Notice that **os_mop::copy_classes()** requires an **os_Set<const os_class_type*>**. In order 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** in the transient schema named **type_name**.

```
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()** will generate 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) ;
}
```

Note that 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, 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)*/
}
```

## The Driver Definition

Here is a driver that creates a graph representing the diagram shown on page 273. 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()**, 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** with name **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** with name **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(member_name) ;

    if (!the_member)
        error("Could not find member with specified name.") ;
```

```
    return *the_member ;
}
```

Driver **main()** function definition

```
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 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 on page 269.

# Chapter 8
# 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. It is editable with tools such as Perl, Awk, and Sed. You can use edited or unedited 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 allows the elimination of most run-time schema lookups during the loading.

Read about the dump/load facility in *ObjectStore Management* Chapter 4, Utilities, osdump: Dumping Databases before you read the discussion in this chapter. This chapter focuses on When Is Customization Required? and addresses the following topics:

- Customizing Dumps on page 289
- Customizing Loads on page 294
- Specializing os_Planning_action on page 295
- Customizing Formatting by Specializing os_Dumper_ specialization on page 299
- Specializing os_Fixup_dumper on page 304
- Specializing os_Fixup_dumper on page 304
- Specializing os_Type_info on page 307
- Specializing os_Type_loader on page 309
- Specializing os_Type_fixup_info on page 316
- Specializing os_Type_fixup_loader on page 318
- **os_Database_table** on page 324
- **os_Dumper_reference** on page 327
- **os_Type_info** on page 331
- **os_Fixup_dumper** on page 333

# 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.

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. For example, if you have a database with objects whose structure depends on the locations of other objects, you might have to customize the dumping and loading of those 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 them.
- Their addresses might hash to different values; that is, for example, **objectstore::get_pointer_numbers()** might return different values for them.

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. Since a dumped collection element hashes to a different value than the corresponding loaded element does, their hash-table slots are different. So the facility 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 form described for ObjectStore collections above.

# 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: 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.

Since 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, and then perform the fixup stage. This means the dumper must dump the arguments for the initialization stage, dump all other objects (except nonroot objects), and 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), and then inserts the elements. And the dumper usually dumps the **create()** arguments, dumps the elements (and all other objects except nonroot objects), and 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 datastructure 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, in order to avoid dumping nonroot portions. Since 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 (like 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 (like 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 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 just 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

In order 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 will serve 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 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 table, following)
- 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 what members of each class 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()** |

# 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 will modify the root object and/or recreate the nonroot portion of the an 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 table, following).
- Implement the members.
- Define an instance.
- Register the instance.

The following table shows what members of each class 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() |

# 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, then, 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, then, 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_table_entry_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, then 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.

## 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 if 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*)**.
- 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&)**.

Here 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()**.
- 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&)**.

Here 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 block(
    planner_entries,
    number_planner_registration_entries,
    __FILE__,
    __LINE__
);
```

This code should be at top level.

# Customizing Formatting by 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 which constructor is called in dumper-generated code for constructing embedded objects

Under some circumstances, you must also 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 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 do not usually have to customize array types, since you can customize the element type of an array type. The default array dumper will call the custom dumper on each array element.

If you do want to customize the array dumper, implement this 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;
```

When you customize the dump and load of a type, you supply code to construct instances of the type during a load — see Implementing create() on page 311. 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*()
- Define the special loader constructor *type*(*type*_data&)

*type*_**data** has 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 class members, the type should be *embedded_class*_ data. All data members corresponding to a base class should have the base class as value type.

All other data members should have the same value type as the coreresponding member of *type*. All data members of *type*_**data** should have the same name as the coreresponding member or base class of *type*.

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.

## 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 base type constructor

## 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()**.
- Retrieve the object being fixed with **os_Fixup_dumper::get_object_to_fix()**. Cast the result to *type*&.
- Output the strings from which the loader will reconstruct the arguments.

To dump arguments that are pointers or C++ references, use the class **os_Dumper_reference**:

- Use **os_Dumper_reference::operator =()** to create a stack-based dumper reference corresponding to the pointer or C++ reference.
- Pass the dumper reference to **operator <<()**, to add its ASCII to the dump stream.

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()** or **os_Dumper_reference::resolve()**.

For example, a fixup form for a collection might include ASCII from which a dumper can reconstruct pointers to 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 *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. See **os_Type_info** on page 331. The derived type must define two members:

- *type*_**info::data**: points to an instance of *type*_**data**, which holds the information required to construct the object being loaded.
- *type*_**info** constructor: takes a *type*_**data** argument; passes other arguments to a base type constructor.

For each instance of *type* being loaded, *type*_**loader::operator ()()** makes an instance of *type*_**data**, as well as 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 will be set by **load()** based on information in the load stream, and 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 class members, the type should be *embedded_class*_data. All data members corresponding to a base class should have the base class as value type.

All other data members should have the same value type as the coreresponding member of *type*. All data members of *type*_**data** should have the same name as the coreresponding member or base class of *type*.

## Implementing the Constructor

```
type_info (
    os_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 info 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 to **operator ()()** as **lstr**.
- Cast to **os_Object_info&** the loader info passed in to **operator ()()**. Pass the result of the cast as **info**.

You can define this constructor to have additional arguments if necessary for your specialization.

# 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 constructor.
- Pass the *type*_**data** to **load()**.
- Pass the *type*_**info** to **create()**.

Here 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);
    create(info);
    return 0;
}
```

## Implementing load()

```
Loader_action* type_loader::load (
    Loader_stream& stream,
    Type_data& given_data
)
```

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.

Once **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.

Here is an example:

```
Loader_action* type_loader::load (
    Loader_stream& stream,
    Type_data& given_data
)
{
    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
);
```

## Implementing create()

**void** *type*_**loader::create (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.
- 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 placement argument.
- In either case, pass *type*_**data** to the *type* constructor (if there is a constructor that takes a *type*_**data&** argument), or call the no-argument *type* constructor.

If you call the no-argument constructor, you must do the following:

- Define **fixup()** to set the values of the new object's data members.

- Define *type*_**dumper::should_use_default_constructor()** to return **1**.

If **create()** rather than **fixup()** sets the data member values, and there are embedded instances of *type*, you must do the following:

- Define the special loader constructor *type*(*type*_**data&**), where *type*_**data** has 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 class members, the type should be *embedded_class*_data. All data members corresponding to a base class should have the base class as value type. All other data members should have the same value type as the coreresponding member of *type*. All data members of *type*_**data** should have the same name as the coreresponding member or base class of *type*..
- Define *type*_**dumper::should_use_default_constructor()** to return **0**.

Since **create()** is not called for embedded objects, either the special loader constructor must exist, or **fixup()** must set all the data members.

The function must also create a mapping record that records the predump and postload locations of the object being loaded:

- Get the type of object being loaded by performing **os_Type_info::get_type()** on **given_info**.
- 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.
- Get the original (that is, predump) location of the loaded object with **os_Type_info::get_original_location()**.
- Retrieve the database table using **os_Database_table::get()**.
- Call **os_Database_table::insert()** on the database table, passing the original location, the dumper reference, and the type of the dumped object.

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**).

Here is an example:

```
void <Type>_loader::create (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();
        Dumper_reference replacing_location(value);
        info.set_replacing_location(value);

        Database_table::get().insert(
            info.get_original_location(),
            replacing_location,
            value_type
        );

    }

    fixup(info.data, value);
}
```

## Implementing fixup()

**void** *type*_**loader::fixup (Type_data& given_data, void* object)**

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 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 **create()** uses the no-argument constructor to construct the object being loaded.

If **create()** does not use the no-argument constructor, and instances of *type* contain no pointers, no C++ references, and no ObjectStore references, you do not have to implement this function. You never have to implement a no-op **fixup()**.

Here is a typical implementation:

```
void type_loader::fixup (Type_data& given_data, void* object)
{
    type_data& data = (type_data&) given_data;
    type& obj = type& *object;
    ...
    /* Fixup pointer. */
    Dumper_reference pointer_location(&obj.pointer);
    Dumper_reference original_referent_location(data.pointer);
    Database_table::get().insert
        (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**.

*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**, as well as 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 will be set by **load()** based on information in the load stream, and 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 info 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 to **operator ()()** as **lstr**.
- Cast to **os_Fixup_info&** the loader info passed in to **operator ()()**. Pass the result of the cast as **info**.

You can define this constructor to have additional arguments if 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 a 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

just 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.

Here is an example:

```
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);
    while( load(stream, data))
        fixup(info);
    return 0;
}
```

This assumes **load()** is implemented to return **0** when the entire info portion has been consumed.

If the info portion is made up of dumped **char***s, allocate the *type*_**Fixup_info** *inside* the load loop:

```
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;
    while( load(stream, data))
        type_fixup_info info(*this, stream, fixup_info, data);
        fixup(info);
    return 0;
}
```

## Implementing load()

```
Loader_action* load (
    Loader_stream& stream,
    Type_data& given_data
);
```

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 of the info portion. If the invocation

operator calls **load()** multiple times**,** each call to **load()** must consume just part of the info portion.

Once **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 just 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***.

Here is an example:

```
Loader_action* type_fixup_loader::load (
    Loader_stream& stream,
    Type_data& given_data
)
{
    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
);
```

## Implementing fixup()

**void fixup (os_Type_fixup_info& info);**

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()** would insert an element in the collection.

The function should

- Cast **given_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.
- Cast the dumper reference to the postload object to *type**.
- Perform fixup on the postload object based on the information in **info**.

Here is an example:

```
void type_fixup_loader::fixup (Type_fixup_info& given_info)
{
    type_fixup_info& info = (type_fixup_info&) info;

    Dumper_reference original_location =
        info.get_original_location();
    if ( ! original_location) {
        // Handle error, there should only be a fixup for an
        // existing object.
    }

    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 block(
    fixup_entries,
    number_fixup_registration_entries,
    __FILE__,
```

```
    __LINE__
);
```

# os_Database_table

The database table records the mapping between predump objects and postload objects, and stores sets of various kinds of fixups, as well as the ignored set. You can obtain the one and only instance of this class with the static member **get()**.

## os_Database_table::get()

**static os_Database_table& get ();**

Returns a reference to the (one and only) database table.

## os_Database_table::insert()

```
void insert (
    const os_Dumper_reference source,
    const os_Dumper_reference target,
    const os_type& referent_type
);
```

Inserts a mapping record that associates a predump object with a postload object. Normally you call this from *type*_**loader::create()**.

**source** is a dumper reference to the predump object.

**target** is a dumper reference to the postload object.

**referent_type** refers to an **os_type** representing the type of the object.

```
void insert (
    os_Reference_fixup_kind::Kind kind,
    const os_Dumper_reference reference,
    const os_Dumper_reference referent_original_location
);
```

Inserts a fixup record that instructs the loader to adjust the pointer, C++ reference, or ObjectStore reference referred to by **reference**. The loader makes the adjustment after loading all object forms and before loading any fixup forms. Normally you call this from *type*_**loader::fixup()**.

**kind** is one of the following:

- **os_Reference_fixup_kind::pointer**
- **os_Reference_fixup_kind::reference_local**
- **os_Reference_fixup_kind::reference_this_db**

- **os_Reference_fixup_kind::reference**
- **os_Reference_fixup_kind::reference_protected_local**
- **os_Reference_fixup_kind::reference_protected**

**reference** refers to the pointer or reference to be fixed up.

**referent_original_location** refers to the predump referent of the pointer or reference to be fixed up.

```
void insert (
    os_segment&,
    const os_Fixup& fixup
);
```

Associates the specified fixup dumper with the specified segment. Each fixup dumper associated with a segment is invoked after all object forms for that segment have been dumped. See Dumper Actions on page 290. Normally you call this function from *type*_**dumper::operator ()()**.

```
void insert (
    os_database&,
    const os_Fixup& fixup
);
```

Associates the specified fixup dumper with the specified database. Each fixup dumper associated with a database is invoked after all object forms for that database have been dumped. See Dumper Actions on page 290. Normally you call this function from *type*_**dumper::operator ()()**.

```
void insert (
    const os_Fixup& fixup
);
```

Associates the specified fixup dumper with the entire dump. Each fixup dumper associated with the entire dump is invoked after all object forms for the dump have been dumped. See Dumper Actions on page 290. Normally you call this function from *type*_**dumper::operator ()()**.

```
void insert (
    const os_Dumper_reference ignored_object
);
```

Inserts the specified object into the *ignored set*. If the object is already in the ignored set, the insertion is silently ignored. Before

dumping an object, the dumper checks to see if the object is in the ignored set. If it is, the dumper does not dump the object.

**ignored_object** is a dumper reference to the object that should not be dumped.

## os_Database_table::find_reference()

```
os_Dumper_reference find_reference (
    const os_Dumper_reference given_reference
) const;
```

Finds the postload object corresponding to a given predump object. Both objects are specified with instances of **os_Dumper_reference**. **given_reference** refers to the predump object. The returned reference refers to the corresponding postload object.

If there is no object that corresponds to the referent of **given_reference**, a null reference is returned.

If **given_reference** is null, a null reference is returned.

See also **os_Database_table::insert()** on page 324 (the first overloading, for mapping records).

## os_Database_table::is_ignored()

```
os_boolean is_ignored (const os_Dumper_reference obj) const;
```

Returns nonzero if **obj** is in the ignored set; returns nonzero otherwise. See **os_Database_table::insert()** on page 324.

# os_Dumper_reference

Instances of the class **os_reference** can be used as substitutes for pointers to predump or postload objects. Given a reference to a predump object, you can retrieve a reference to the corresponding postload object (using **os_Database_table::find_reference()**). Dumper references are required as arguments to certain functions your specializations call, such as **os_Database_table::insert()**.

As with a pointer, once the object referred to by a dumper reference is deleted, use of the reference accesses arbitrary data and might cause a segmentation violation.

You can construct or set a reference with **os_Dumper_reference::os_Dumper_reference()** or **os_Dumper_reference::operator =()**. You can resolve a reference with **os_Dumper_reference::resolve()** or **os_Dumper_reference::resolve()**.

## os_Dumper_reference::operator void*()

**operator void*() const;**

Returns the pointer for which the specified reference is a substitute.

## os_Dumper_reference::operator =()

**os_Dumper_reference& operator = (const os_Dumper_reference&);**

Establishes the referent of the right operand as the referent of the left operand.

**os_Dumper_reference& operator = (void* object);**

Establishes the object pointed to by the right operand as the referent of the left operand.

## os_Dumper_reference::os_Dumper_reference()

**os_Dumper_reference (const void*);**

Constructs a reference to substitute for the specified **void***.

```
os_Dumper_reference (
    os_unsigned_int32 database_number,
    os_unsigned_int32 segment_number,
    os_unsigned_int32 offset
);
```

Constructs a reference to the object with the specified database number, segment number, and offset.

**os_Dumper_reference (const os_Dumper_reference&);**

Constructs a reference with the same referent as the specified reference.

**os_Dumper_reference ();**

Constructs a null reference, that is, a reference without a current referent. See **os_Dumper_reference::operator =()** on page 327.

## os_Dumper_reference::resolve()

**void* resolve() const;**

Returns the valid **void*** for which the specified reference is a substitute.

## os_Dumper_reference::operator ==()

**os_boolean operator == (const os_Dumper_reference&) const;**

Returns **1** if the arguments have the same referent; returns **0** otherwise.

## os_Dumper_reference::operator <()

**os_boolean operator < (const os_Dumper_reference&) const;**

If the first argument and second argument refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument precedes the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

## os_Dumper_reference::operator >()

**os_boolean operator > (const os_Dumper_reference&) const;**

If the first argument and second argument refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument follows the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

## os_Dumper_reference::operator !=()

**os_boolean operator != (const os_Dumper_reference&) const;**

Returns **1** if the arguments have different referents; returns **0** otherwise.

## os_Dumper_reference::operator >=()

**os_boolean operator >= (const os_Dumper_reference&) const;**

If the first argument and second argument refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument follows or is the same as the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

## os_Dumper_reference::operator <=()

**os_boolean operator <= (const os_Dumper_reference&) const;**

If the first argument and second argument refer to elements of the same array or one beyond the end of the array, a return value of **1** indicates that the referent of the first argument precedes or is the same as the referent of the second, and a return value of **0** indicates that it does not. Otherwise the results are undefined.

## os_Dumper_reference::operator !()

**os_boolean operator ! () const;**

Returns nonzero if the reference is null, that is, has no current referent.

## os_Dumper_reference::get_database()

**os_database* get_database () const;**

Returns the database containing the referent of this reference.

## os_Dumper_reference::get_database_number()

**os_unsigned_int32 get_database_number () const;**

Returns the database table number of the database containing the referent of this reference.

## os_Dumper_reference::get_segment()

**os_segment* get_segment () const;**

Returns the segment containing the referent of this reference.

## os_Dumper_reference::get_segment_number()

**os_unsigned_int32 get_segment_number () const;**

Returns the segment number of the segment containing the referent of this reference.

## os_Dumper_reference::get_offset()

**os_unsigned_int32 get_offset () const;**

Returns the offset of the referent of this reference within its segment.

## os_Dumper_reference::get_string()

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

Returns the reference's value as an encoded string.

## os_Dumper_reference::is_valid()

**os_boolean is_valid () const;**

Returns nonzero if **this** is completely valid; returns **0** otherwise.

# os_Type_info

Instances of this class hold information about the loading of the object form or fixup form currently being processed. It defines members for getting the type and predump location of the object being loaded or fixed up, as well as members for getting and setting the postload location of the object.

## os_Type_info::os_Type_info()

```
os_Type_info (
    os_Type_loader&,
    os_Loader_stream&,
    os_Object_info&
);
```

Constructs an object to hold information about the object currently being loaded by a specified loader. You must also pass as constructor argument the info for the previous object load. See Implementing operator ()() on page 309 in the section on specializing **os_Type_loader**.

## os_Type_info::get_original_location()

```
os_Dumper_reference get_original_location () const;
```

Returns a reference that resolves to the predump location of the object being loaded. See Implementing create() on page 311 in the section on specializing **os_Type_loader**.

## os_Type_info::get_replacing_location()

```
os_Dumper_reference get_replacing_location () const;
```

Returns a reference that resolves to the postload location of the object being loaded. See Implementing create() on page 311 in the section on specializing **os_Type_loader**.

## os_Type_info::set_replacing_location()

```
void set_replacing_location (os_Dumper_reference location);
```

Records the location at which the postload object has been created. See Implementing create() on page 311 in the section on specializing **os_Type_loader**.

## os_Type_info::get_type()

**const os_type& get_type () const;**

Returns a reference to an **os_type** that represents the type of the object being loaded. See Implementing create() on page 311 in the section on specializing **os_Type_loader**.

## os_Type_info::get_replacing_segment()

**os_segment& get_replacing_segment () const;**

Returns a reference to the postload segment of the object being loaded.

## os_Type_info::get_replacing_database()

**os_database& get_replacing_database () const;**

Returns a reference to the postload database of the object being loaded.

# os_Fixup_dumper

This class serves as base class for your customized fixup-form dumpers. See Specializing os_Fixup_dumper on page 304. Some of the members of your specialization might have to call functions defined by the base class, **os_Fixup_dumper**. These functions are described here.

## os_Fixup_dumper::os_Fixup_dumper()

```
os_Fixup_dumper (
    os_Dumper_stream&,
    os_Dumper&,
    const os_class_type&,
    const os_Dumper_reference object_to_fix,
    unsigned number_elements = 0
);
```

Constructs a fixup dumper. The constructors for your customized fixup dumpers pass arguments to this constructor.

```
os_Fixup_dumper (const os_Fixup_dumper&);
```

Copy constructor.

## os_Fixup_dumper::get_object_to_fix()

```
os_Dumper_reference get_object_to_fix () const;
```

Returns a dumper reference to the object for which **this** dumps a fixup form. See Implementing dump_info() on page 304 in the section on specializing **os_Fixup_dumper**.

## os_Fixup_dumper::get_type()

```
os_type &get_type() const;
```

Returns a reference to an **os_type** that represents the type of the object for which **this** is a fixup dumper. See Implementing dump_info() on page 304 in the section on specializing **os_Fixup_dumper**.

## os_Fixup_dumper::~os_Fixup()

```
virtual ~os_Fixup ();
```

Virtual destructor

## os_Fixup_dumper::get_number_elements()

```
unsigned get_number_elements () const;
```

If this is a fixup for an array, returns the number of elements to be fixed up. Returns **0** otherwise.

# Chapter 9
# Advanced Schema Evolution

This chapter provides information about the ObjectStore schema evolution facility. For a basic understanding of tasks you must perform to complete a schema evolution project, see Chapter 8, Schema Evolution, in the *ObjectStore 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 database(s) 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 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*. See Instance Initialization on page 338.
- *Instance transformation*. See Instance Transformation on page 342.

# 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 where a field has changed type, and the old and new types are assignment compatible, the new field is initialized by assignment from the old field value.

The initialization rules are discussed in Instance Initialization Rules on page 384.

## Pointers to Modified Objects and Their Subobjects

During the initialization phase, the address of an instance being migrated generally changes. The reason for this is that migration actually consists of making a copy of the old unmigrated instance, and then modifying this copy. The copy and the old instance will be in the same segment, but their offsets within the segment will be different.

Because of this, the schema evolution facility automatically modifies all pointers to the instance so that they point to the new modified instance. This is done for *all* pointers in the databases being evolved, including pointers contained in instances of unmodified classes, cross-database pointers, and pointers to subobjects of migrated instances.

## Illegal Pointers

During this process of adjusting pointers to modified instances, ObjectStore might detect various kinds of illegal pointers. For example, it might detect a pointer to the value of a data member that has been removed in the new schema. Since the data member has been removed, the subobject serving as the value of that data member is deleted as part of instance initialization. Any pointer to such a deleted subobject is illegal, and is detected by ObjectStore.

In such a case, you can provide a special handler function to process the illegal pointer (for example, by changing it to null or simply reporting its presence). Each time an illegal pointer is detected, the handler function is executed on the pointer, and then schema evolution is resumed. If you do not provide a handler function, an exception is signaled by default when an illegal pointer is encountered. If you want, you can specify that illegal pointers be ignored.

## C++ References

C++ references are treated as a kind of pointer. References to migrated instances are adjusted just as described above. Illegal 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. You are given an option concerning local references. Recall that to resolve a local reference you must specify the database containing the referent. If you want, you can direct ObjectStore to resolve each local reference using the database in which the reference itself resides. If you do not use this option, local references will not be adjusted during instance initialization (but you can provide a transformer function so that they are adjusted during the *instance transformation* phase; see Instance Transformation on page 342).

As with pointers, you can supply handler functions for illegal references. If you do not supply an illegal reference handler, evolution continues uninterrupted when an illegal reference is encountered. The reference is left unmodified and no exception is signaled.

Illegal pointers and references, and illegal pointer and reference handlers, are described in detail in Illegal Pointers on page 373.

## 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 internally marks the query so that subsequent attempts to use it cause a run-time error.

As with illegal pointers, you can handle obsolete queries or indexes by providing a special handler function for each. A dropped index handler, for example, might create a new index using a path that is legal under the new schema. If you do not supply handlers, ObjectStore signals an exception when an obsolete query or index is encountered.

Handlers for dropped indexes and obsolete queries are discussed in Obsolete Index and Query Handlers on page 381.

## Instance Reclassification

The schema evolution facility allows for one special form of instance migration, which allows you to *reclassify* instances of a given class as instances of a class derived from the given class. This form of migration is special because it is not, strictly speaking, a case of modifying instances to conform to a new class definition. However, instance reclassification is typically desirable when new subclasses are added to a schema. Instances of the base class can be given a more specialized representation by being classified as instances of one of the derived classes.

Reclassification occurs during the initialization phase. You specify how instances of a given base class are to be reclassified by associating a *reclassification function* with the base class. This function takes an instance of the base class as its argument, and returns the name of the instance's new class, if it is to be reclassified.

Reclassified instances can then be transformed during the transformation phase, as with any migrated instances. A reclassified instance will be transformed by the transformer function associated with its new class, a class derived from its original class.

Instance reclassification is discussed in Instance Reclassification on page 366.

## 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 how the migrated instances of each modified class will 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 382.

# Instance Transformation

For some schema changes, the instance initialization phase is all that is needed. But in other cases, further modification of class instances or associated data structures is required to complete the schema evolution. This further modification is generally application dependent, so ObjectStore allows you to define your own functions, *transformer functions*, to perform the task.

## 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 particularly useful when you want to set the value of some field of a migrated instance based on the values of some field or fields of the corresponding old instance. For this purpose, the evolution facility provides a function that allows you to retrieve the old instance corresponding to a given new instance.

You can also use a transformer function to adjust local references (see Instance Initialization Rules on page 384). A transformer associated with a class containing an **os_reference_local** or **os_reference_protected_local** could perform the adjustment by retrieving the new version of each local reference's referent, and assigning it to the reference.

In addition, transformers are useful for updating data structures that depend on the addresses of migrated instances. A hash table, for example, that hashes on addresses should be rebuilt using a transformer. Note that you do *not* need to rebuild a data structure if the position of an entry in the structure does *not* depend on the address of an object pointed to by the entry, but depends instead, for example, on the value of some field of the object pointed to. Such data structures will still be correct after the instance initialization phase.

Once the transformation phase is complete, all the old unmigrated instances are deleted. (If the old instances of a given class are not needed for the transformation phase, you can direct ObjectStore to delete them during the initialization phase. See Recycling Old Storage on page 351.)

Using transformers is discussed in Using Transformer Functions on page 350.

# Initiating Evolution with **evolve()**

To perform schema evolution, you make and execute an application that invokes the static member function **os_schema_evolution::evolve()**. The function must be called outside the dynamic scope of a transaction. The application must include the header file **ostore/schmevol.hh** and link with the libraries **libosse.a**, **liboscol.a**, and **libos.a**.

The function **evolve()** 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

## Databases to Evolve

You specify the database or databases to be evolved as the second argument to **evolve()**. If you are evolving just a single database, you supply a **char***, the pathname of the database. If you are evolving more than one database, you supply an **os_collection&** or **os_Collection<char*>&**, a set containing the databases' pathnames.

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()**. So 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()**.

## 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 illegal pointer; see Illegal Pointers on page 373).

When evolution is interrupted, the work database records a consistent intermediate state of the evolution process.

Subsequently calling **evolve()** using the same work database will cause evolution to be resumed from the point of interruption.

After evolution successfully completes, you should delete the work database.

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.

## Resolution of Local References

As mentioned earlier, you are given an option regarding the resolution, during evolution, of local ObjectStore references. (Recall that the referent's database must be specified for resolution of local references.) If you call the static member function **os_schema_evolution::set_local_references_are_db_relative()**, supplying a nonzero **int** (true) as argument, local references will be resolved using the database in which the reference itself resides. Otherwise local references will not be adjusted during the instance initialization phase (see Illegal Pointers on page 373).

# Example: Changing the Value Type of a Data Member

Consider an example that involves changing the value type of a data member.

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();
}
```

Here, the value type of the data member **part_id** has changed from **short** to **long**. The constructor's argument type has also changed. Since C++ provides a standard conversion from **short** to **long**, migrated instances of the class **part** will 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 this:

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

#include "part1new.hh" /* the new definition */

main() {

    objectstore::initialize();

    os_schema_evolution::evolve(
        "/example/workdb",
        "/example/partsdb"
    );
}
```

Note that the header file **ostore/schmevol.hh** is included.

Here, 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 this:

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

#include "part1new.hh" /* the new definition */

main() {

    objectstore::initialize();

    os_collection::initialize();

    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
    );
}
```

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 */

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

The instance migration phase of the schema evolution process will migrate 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 will also initialize 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 here. 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 like 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/ex1.comp_schema_db /example/partsdb
```

For information on the **ossevol** utility, see Schema Evolution with ossevol in Chapter 8 of the *ObjectStore C++ API User Guide*.

# Using Transformer Functions

The instance initialization phase leaves migrated instances in a well-defined state. But 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, and you then associate the function with the class on whose instances the function will operate (see Associating a Transformer with a Class on page 351).

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 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 that has already undergone instance initialization.

Form of the call

```
void my_transform_function(void *the_new_obj)
```

Transformer functions frequently perform processing that is based on the state of the old unevolved object corresponding to the object being operated on. The evolution facility provides a means of accessing the old object. This is discussed in the next section, Accessing Unevolved Objects on page 353.

## Associating a Transformer with a Class

With the transform function defined, you can associate the function with a class and invoke the evolution process. The association is made by calling the static member function **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()**.

**augment_post_evol_transformers()** function

The function **augment_post_evol_transformers()** has the following 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*>&
   );
```

**os_transformer_binding()** function

You can construct an instance of **os_transformer_binding** by supplying a class name and a function pointer as arguments to the constructor, as in

```
os_transformer_binding("part", part_transform)
```

So a typical call to **augment_post_evol_transformers()** would be

```
os_schema_evolution::augment_post_evol_transformers (
   os_transformer_binding("part", part_transform)
);
```

## Recycling Old Storage

For classes whose instances' old state does not need to be accessed by any transformer, and for removed classes, you can increase space efficiency during the evolution process by having their old unevolved instances deleted during the instance initialization phase, allowing their space to be used for new instances. You do this with one call to **os_schema_evolution::augment_classes_to_be_recycled()** for each class whose old instances can be deleted.

**augment_classes_to_be_recycled()** function

This function is declared as follows:

```
static void
   os_schema_evolution::augment_classes_to_be_recycled(
      const char *name_of_class_to_be_recycled
   );
```

The calls should precede the call to **evolve()**.

You can also call this function once for all the classes to be recycled, if you pass an **os_Collection<char*>** containing the names of all the classes to be recycled. In this case you use the overloading

```
static void
    os_schema_evolution::augment_classes_to_be_recycled(
        const os_Collection<char*>
        &names_of_classes_to_be_recycled
    );
```

Again, this call should precede the call to **os_schema_evolution::evolve()**.

Note that the old unevolved instances of each modified class are deleted following completion of the transformation phase, whether or not you have specified the class as one to be recycled.

# Accessing Unevolved Objects

Transformer functions (see Using Transformer Functions on page 350), as well as reclassification functions (see Instance Reclassification on page 366), often perform processing that is based on the state of the old unevolved object corresponding to the object being operated on. This section tells you how to access that state.

Given a pointer, **the_new_obj**, to an initialized object, retrieving a data member value for the corresponding old unevolved object has the following steps:

1 Retrieve an **os_typed_pointer_void** that refers to the old object. An **os_typed_pointer_void** is a special container object that encapsulates a **void*** pointer to the old instance and an object representing the instance's type.

2 Retrieve a **void*** pointer to the old object.

3 Retrieve a pointer to the object representing the type of the old object.

4 Given the type object, retrieve a pointer to the object representing the data member whose value you want to access.

5 Given the old object and the data member object, retrieve the old data member value.

These steps are necessary because the new schema provides the type universe for transformer and reclassification functions. The old class definitions are not part of a transformer's schema, and therefore you cannot use the usual member access notation, **.member-name**, to access fields of the old instance.

### Retrieving **os_typed_pointer_void** and **void*** pointers

You can retrieve an **os_typed_pointer_void** to the old unevolved instance using the static member function **os_schema_evolution::get_unevolved_object()**. To retrieve the pointer itself you simply assign the **os_typed_pointer_void** to a **void*** variable, as in

```
os_typed_pointer_void old_obj_typed_ptr =
   os_schema_evolution::get_unevolved_object(a_new_obj);
void *an_old_obj = old_obj_typed_ptr;
```

This works because the class **os_typed_pointer_void** defines **operator void*()** to return the pointer.

### Retrieving the type and the data member

You can retrieve the type with the member function **os_typed_pointer_void::get_type()**.

```
const os_class_type &c = old_obj_typed_ptr.get_type();
```

You retrieve a pointer to the object representing the data member of a specified name defined by a specified type using **os_class_type::find_member()**.

### Retrieving the data member value

Finally, you retrieve the value of a specified data member for a specified object using **os_fetch()**:

```
os_fetch(the_old_obj, *c.find_member("part_id"), the_old_val);
```

As mentioned earlier, the instance initialization phase of evolution automatically modifies all pointers to instances of modified classes so that they reference the new migrated instances. This is true even for pointers contained in old unmigrated instances. So if you access an old data member during the instance transformation phase, and the value of the member is a pointer to an instance of a class that was also modified, the value you retrieve will point to the new migrated instance (see Example: Changing Inheritance on page 360).

### Functions used to access unevolved objects

Here are the declarations of the functions used to access unevolved objects:

```
static os_typed_pointer_void os_schema_evolution::
    get_unevolved_object(void *new_obj);

os_typed_pointer_void::operator void*() const;

const os_type &os_typed_pointer_void::get_type() const;

const os_member *os_class_type::
    find_member(const char *name) const;
```

There is also a function for retrieving the address of the new version of a specified unevolved object, **get_evolved_object()**.

### **os_fetch()** global function

The global function **os_fetch()** has an overloading for each built-in C++ type:

```
void *os_fetch(
    const void *p, const os_member_variable&, void *&value);

unsigned long os_fetch(
    const void *p, const os_member_variable&,
    unsigned long &value);

long os_fetch(
    const void *p, const os_member_variable&, long &value);
```

```
unsigned int os_fetch(
   const void *p, const os_member_variable&,
   unsigned int &value);

int os_fetch(
   const void *p, const os_member_variable&, int &value);

unsigned short os_fetch(
   const void *p, const os_member_variable&,
   unsigned short &value);

short os_fetch(
   const void *p, const os_member_variable&, short &value);

unsigned char os_fetch(
      const void *p, const os_member_variable&,
      unsigned char &value);

char os_fetch(
   const void *p, const os_member_variable&, char &value);

float os_fetch(
   const void *p, const os_member_variable&, float &value);

double os_fetch(
   const void *p, const os_member_variable&, double &value);

long double os_fetch(
   const void *p, const os_member_variable&,
   long double &value);
```

### **os_store()** global function

Once you have retrieved an old data member value, you can usually just assign it to the new data member. But if the value type of the new data member is a **const** or reference type, you should use **os_store()** to set the new member value.

```
os_store(the_new_obj, c.find_member("part_id"), the_old_val);
```

Like **os_fetch()**, **os_store()** has an overloading for each built-in C++ type:

```
void os_store(
   void *p, const os_member_variable&, const void *value);

void os_store(
   void *p, const os_member_variable&,
   const unsigned long value);

void os_store(
   void *p, const os_member_variable&, const long value);

void os_store(
   void *p, const os_member_variable&,
   const unsigned int value);
```

```
void os_store(
    void *p, const os_member_variable&, const int value);

void os_store(
    void *p, const os_member_variable&,
    const unsigned short value);

void os_store(
     void *p, const os_member_variable&, const short value);

void os_store(
    void *p, const os_member_variable&,
    const unsigned char value);

void os_store(
     void *p, const os_member_variable&, const char value);

void os_store(
    void *p, const os_member_variable&, const float value);

void os_store(
     void *p, const os_member_variable&, const double value);

void os_store(
    void *p, const os_member_variable&,
    const long double value);
```

**os_fetch_address()** global function

You can get the address of a data member value with **os_fetch_address()**, declared

```
void *os_fetch_address(void *p, const os_member_variable&);
```

**os_member_variable::get_type()** function

And you can get the value type of a data member with **os_member_variable::get_type()**, declared

```
const os_type &os_member_variable::get_type() const;
```

Together with **os_fetch()**, these functions allow you to access not only data members, but also data members of data member values, and so on.

Accessing an inherited data member

To access an inherited data member, the following functions are useful:

```
const os_base_class &os_class_type::find_base_class(
    char *base_class_name) const;

void *os_fetch_address(
    void *p, const os_base_class_variable&);

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

Header file requirement

To use the functions described in this section, you must include the header file **<ostore/mop.hh>**.

# Example: Using Transformers

Now consider an example that uses a transformer function.

Suppose that instead of changing the value type of the class **part** (see the previous example) from **short** to **long**, you want to change it from **short** to **char***, so arbitrary strings can be used for part IDs:

Existing **part** class definition

```
class part {
    public:
        short part_id;
        part(short id) { part_id = id; }
        static os_typespec *get_os_typespec();
}
```

New **part** class definition

And you want to change the definition to be as follows:

```
class part {
    public:
        char *part_id;
        part(char *id) {
            int len = strlen(id) + 1;
            part_id = new(
                os_segment::of(this),
                os_typespec::get_char(),
                len
            ) char[len];
            strcpy(part_id, id);
        }
        static os_typespec *get_os_typespec();
}
```

Since there is no standard C++ conversion from **short** to **char***, the new field will be initialized to **(char*) (0)** during the instance initialization phase of schema evolution. But we can direct the evolution facility to overwrite this initialization during the transformation phase, and establish a new **part_id** value for a migrated instance based on the value of **part_id** for the corresponding unmigrated instance.

To do this, supply a transformer function and associate it with the class **part**. As part of the instance migration process, the ObjectStore schema evolution facility will invoke this transformer function on each instance of the class.

**part_transform()** transformer function

Here is how such a transformer function might be defined.

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

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

#include <stdio.h>
#include <string.h>

#include "part2new.hh"

static void part_transform(void *the_new_obj) {

    /* get a typed ptr to the old obj */
    os_typed_pointer_void old_obj_typed_ptr =
        os_schema_evolution::get_unevolved_object(
            the_new_obj);

    /* get a void* ptr to the old obj; implicit operator void*() call */
    void *the_old_obj = old_obj_typed_ptr;

    /* get the type of the old obj */
    const os_class_type &c = old_obj_typed_ptr.get_type();

    /*  get the old data member value */
    int the_old_val;
    os_fetch(the_old_obj, *c.find_member("part_id"),
            the_old_val);

    /*  convert the old value to string form */
    char conv_buf[16];
    sprintf(conv_buf, "%d", the_old_val);

    int len = strlen(conv_buf) + 1;
    part *part_ptr = (part *)the_new_obj;
    part_ptr->part_id =
        new(os_segment::of(the_new_obj),
            os_typespec::get_char(), len) char[len];
    strcpy(part_ptr->part_id, conv_buf);

}
```

This function, **part_transform()**, sets the value of **part_id** in the new instance to the string denoting the integer value of **part_id** in the old unevolved instance. So, for example, if the old **part_id** was the integer **1138**, the transformer sets the new **part_id** to a pointer to the character array **1138**.

With the transform function defined, you can associate the function with the class **part** and invoke the evolution process. As mentioned above, the association is made using a function call from within the application that invokes schema evolution.

### **main()** function

The **main()** function associates **part_transform()** with the class **part** by creating an **os_transformer_binding** for the function and the class, and invoking **augment_post_evol_transformers()** on it.

Once the association between transformer and class is made, evolution is invoked.

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

#include "part2new.hh"

main() {

    objectstore::initialize();

    /*  associate part_transform() with the class part */
    os_schema_evolution::augment_post_evol_transformers(
        os_transformer_binding("part", part_transform)
    );

    /*  initiate evolution */
    os_schema_evolution::evolve(
        "example/workdb", "example/partsdb"
    );
}
```

Note that if the class **part** has classes derived from it, the instances of these derived classes must also be migrated, since each instance of the derived classes has a subobject corresponding to the base class **part**. The transformer **part_transform()** is run on these subobjects as well.

# Example: Changing Inheritance

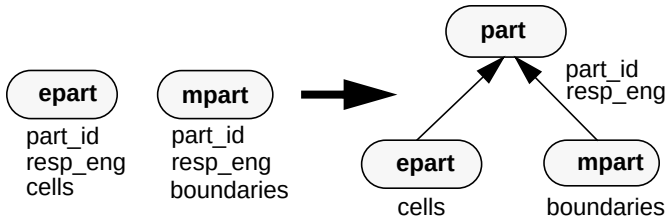
Here is an example that involves deleting some data members from a class, as well as changing the class to inherit from a new base class.

Consider a database schema that uses the classes **epart**, for electrical part, and **mpart**, for mechanical part, and suppose these classes both have data members for **part_id** and **responsible_engineer**. The example below shows how to add a common base class, **part**, to these two classes, and 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 merely on sameness of name, since 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.

Here are the old 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;
         brep =0;
      }
};
```

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) { brep =0; }
};
```

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

/* these contain the new definitions */
#include "part.hh"
#include "new_epart.hh"
#include "new_mpart.hh"

static void dummy() {

    OS_MARK_SCHEMA_TYPE(epart);
    OS_MARK_SCHEMA_TYPE(mpart);
    OS_MARK_SCHEMA_TYPE(part);
    . . .

}
```

The instance migration phase of the schema evolution process modifies the instances of **epart** and **mpart** by eliminating the **part_id** and **responsible_engineer** fields from the subobject corresponding to the derived class. It also adds 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

Suppose you want 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.

To do this, you supply a transformer function for each derived class, **epart** and **mpart**.

```
static void epart_transform(void *the_new_obj) {

    /*  get a typed ptr to the old instance */
    os_typed_pointer_void old_obj_typed_ptr =
        os_schema_evolution::get_unevolved_object(
            the_new_obj
        );

    /* get a void* ptr to the old obj */
    void *the_old_obj = old_obj_typed_ptr;

    /* get the type of the old obj */
    os_class_type &c = old_obj_typed_ptr.get_type();

    /* get the old data member values */
    int the_old_id_val;
    os_fetch(
        the_old_obj,
        *c.find_member("part_id"),
```

```
        the_old_id_val
    );

    void *the_old_resp_eng_val;
    os_fetch(
        the_old_obj,
        *c.find_member("responsible_engineer"),
        the_old_resp_eng_val
    );

    /* set the new data member values */
    epart *epart_ptr = (epart*)the_new_obj
    epart_ptr->part_id = the_old_id_val;
    epart_ptr->responsible_engineer =
        (employee*)the_old_resp_eng_val;
}

static void mpart_transform(void *the_new_obj) {

    /*  get a typed ptr to the old instance */
    os_typed_pointer_void old_obj_typed_ptr =
        os_schema_evolution::get_unevolved_object(
            the_new_obj
        );

    /*  get a void* ptr to the old obj */
    void *the_old_obj = old_obj_typed_ptr;

    /*  get the type of the old obj */
    os_class_type &c = old_obj_typed_ptr.get_type();

    /*  get the old data member values */
    int the_old_id_val;
    os_fetch(
        the_old_obj,
        *c.find_member("part_id"),
        the_old_id_val
    );

    void *the_old_resp_eng_val;
    os_fetch(
        the_old_obj,
        *c.find_member("responsible_engineer"),
        the_old_resp_eng_val
    );

    /*  set the new data member values */
    mpart *mpart_ptr = (mpart*)the_new_obj;
    mpart_ptr->part_id = the_old_id_val;
    mpart_ptr->responsible_engineer =
        (employee*)the_old_resp_eng_val;

}
```

Here, 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.

Note that, if the current evolution calls for the migration of instances of the class **employee**, the value of **responsible_engineer** retrieved from the old instance will be 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, since we are usually interested in the evolved version of the old data member value.

### Example: associating transformers with their classes and invoking evolution

Here is an application that associates the transformers with their classes and invokes evolution.

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

#include "part.hh"
#include "new_epart.hh"
#include "new_mpart.hh"

main() {
    objectstore::initialize();

    /* associate epart_transform() with the class epart */
    os_schema_evolution::augment_post_evol_transformers(
        os_user_tranformer_binding("epart", epart_transform)
    );

    /*  associate mpart_transform() with the class mpart */
    os_schema_evolution::augment_post_evol_transformers(
        os_user_tranformer_binding("mpart", mpart_transform)
    );

    /* perform the evolution process */
    os_schema_evolution::evolve(
        "/example/workdb", "/example/partsdb"
    );

}
```

For databases undergoing the evolution described in this example, ObjectStore detects as illegal any pointers to **epart**s or **mpart**s 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** (since 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 illegal pointers, see Illegal Pointers on page 373.

If the example is modified to include a leftmost base class for **epart** and **mpart**, both before and after evolution, **void*** pointers to **epart**s and **mpart**s will not be illegal.

# Instance Reclassification

As described above, the ObjectStore schema evolution facility allows you to migrate an instance to a subclass of its original class. This is particularly useful when new derived classes that are more appropriate classes for existing instances of the base class are added to a schema.

To reclassify an instance, you must define a reclassification function and associate it with the class whose instances are to be reclassified. As part of the instance initialization phase of schema evolution, ObjectStore will execute the reclassification function on each instance of the function's associated class and reclassify the instance according to the return value of the function.

## Signature of Reclassification Functions

Reclassifiers are static functions with a return type of **char*** and one argument of type **os_typed_pointer_void&** (see Using Transformer Functions on page 350). This argument is a reference to a typed pointer to the object to be reclassified, an unevolved instance of the original class.

```
static char *my_reclassification_function(
    os_typed_pointer_void &old_obj_typed_ptr
);
```

The return value, for a given instance, should be a string naming the new class the instance is to have. If the return value is **0**, the instance will retain its current type.

As with transformers, the schema for reclassification functions is the new schema. So to access fields of the object being reclassified, you must use **os_typed_pointer_void::get_type()**, **os_class_type::find_member()**, and **os_fetch()**. See Using Transformer Functions on page 350 and the example in Example: Reclassifying Instances on page 368.

## Associating a Reclassifier with a Class

With the reclassification function defined, you can associate the function with a class and invoke the evolution process. You make the association by calling the static member function **os_schema_evolution::augment_subtype_selectors()** in the application

performing evolution. The call should be made before the call to **evolve()**.

### **augment_subtype_selectors()** function

The function **augment_subtype_selectors()** takes an instance of **os_evolve_subtype_fun_binding** as argument. You can construct an instance of this class by supplying a class name and a function pointer as arguments to the constructor, as in

```
os_evol_subtype_fun_binding("part", part_reclassifier)
```

So a typical call to **augment_subtype_selectors()** would be

```
os_schema_evolution::augment_subtype_selectors (
    os_evolve_subtype_fun_binding("part", part_reclassifier)
);
```

# Example: Reclassifying Instances

Consider a schema containing the class **part** with data members **cells** (a pointer to the collection of subcircuits of an electrical part) and **boundary_rep** (a pointer to the geometric representation of the boundary of a mechanical part). Suppose that the **part**s that have a nonnull value for **cells** have **0** for **boundary_rep**, and the parts that have a nonnull value for **boundary_rep** have **0** for **cells**.

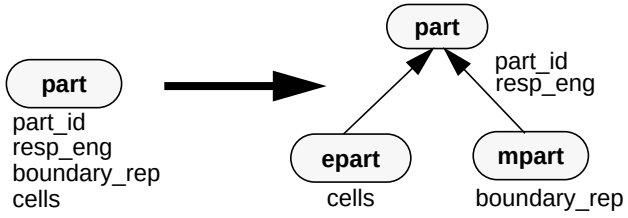
In such a case, it might be desirable to modify this schema to include two new classes derived from **part**, **epart** (for electrical part) and **mpart** (for mechanical part). The data member **cells** can be moved out of **part** and into **epart**, and the member **boundary_rep** can be moved out of **part** and into **mpart**.

In addition to adding the subclasses to the schema, we should migrate existing instances of **part** so that those with a nonnull value for **cells** are reclassified as **epart**s, and those with a nonnull value for **boundary_rep** are reclassified as **mpart**s.

The schema change in this example involves

- Deleting the members **part::cells** and **part::boundary_rep**
- Deriving two new classes, **epart** and **mpart**, from **part**

Moving data members of **part** to new subtypes




Again, note that moving a data member from supertype to subtype should be viewed as deletion of the data member from the supertype, together with addition of a new, distinct data member to the subtype.

Existing **part** class definition

Here is the original definition of the class **part**:

```
class part {
  public:
    int part_id;
```

```
        employee *responsible_engineer;
        os_Collection<cell*> *cells;
        brep *boundary_rep;
        part(int id, employee *eng) {
            part_id = i;
            responsible_engineer = eng;
            boundary_rep = 0;
        }
};
```

### New class definitions

Here are the class definitions of the new schema:

```
class part {
    public:
        int part_id;
        employee *responsible_engineer;
        part(int id, employee *eng) {
            part_id = i;
            responsible_engineer = eng;
        }
};

class epart : public part {
    public:
        os_Collection<cell*> *cells;
        . . .
        epart(int i) : part(i) { cells = 0; }
}

class mpart : public part {
    public:
        brep *boundary_rep;
        . . .
        mpart(int i) : part(i) { brep = 0; }
};
```

### Schema source file

The schema source file for this executable should contain the definitions of **epart** and **mpart**, as well as the new definition of **part**.

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

/* these contain the new definitions */
#include "new_part.hh"
#include "epart.hh"
#include "mpart.hh"

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

The instance migration phase of the schema evolution process will modify the instances of **part** by eliminating the **cells** and **boundary_rep** fields. But first, you would like each **part** to be reclassified according to whether it uses the **cells** field or the **boundary_rep** field.

### Reclassification function

To do this, you define a reclassification function and associate it with the class **part**. Here is the function definition:

```
static char *part_reclassifier(
        os_typed_pointer_void &old_obj_typed_ptr
) {
    /* get a void* ptr to the old obj */
    void *the_old_obj = old_obj_typed_ptr;

    /* get the type of the old obj */
    os_class_type &c = old_obj_typed_ptr.get_type();

    /* get the old cells value */
    os_Collection<cell*> *the_old_cells_val;
    os_fetch(
        the_old_obj,
        *c.find_member("cells"),
        the_old_cells_val
    );

    if (the_old_cells_val)
        return "epart"; /*  make it an epart */

    /* get the old boundary_rep value */
    brep *the_old_boundary_rep_val;
    os_fetch(
        the_old_obj,
        *c.find_member("boundary_rep"),
        the_old_boundary_rep_val
    );

    if (the_old_boundary_rep_val)
        return "mpart"; /*  make it an mpart */

    return 0; /*  leave it alone */
}
```

The reclassification of each **part** essentially amounts to supplementing it with a subobject corresponding to the derived class, **epart** or **mpart**. The subobject is initialized as if by a constructor that initializes each member to **0**. We can overwrite this initialization by defining transformer functions for the derived classes.

Note that the reclassification function is associated with the *original* class (the base class) of the instances it operates on, while the transformer functions (see below) are associated with the *new* classes (the derived classes) of the instances they operate on.

### Transformer functions

Here are the transformer functions that allow you to set the values of **cells** and **boundary_rep** for the new instances according to their values in the old instances.

```
static void epart_transform(void *the_new_obj) {

    /* get a typed ptr to the old instance */
    os_typed_pointer_void old_obj_typed_ptr =
        os_schema_evolution::get_unevolved_object(
        the_new_obj);

    /* get a void* ptr to the old obj */
    void *the_old_obj = old_obj_typed_ptr;

    /* get the type of the old obj */
    os_class_type &c = old_obj_typed_ptr.get_type();

    /* get the old data member values */
    os_Collection<cells*> the_old_cells_ val;
    os_fetch(the_old_obj,*c.find_member("cells"),
        the_old_cells_val);

    /*  set the new data member value */
    the_new_obj->cells = the_old_cells_val;
}

static void mpart_transform(void *the_new_obj) {

    /*  get a typed ptr to the old instance */
    os_typed_pointer_void old_obj_typed_ptr =
        os_schema_evolution::get_unevolved_object(
        the_new_obj);

    /*  get a void* ptr to the old obj */
    void *the_old_obj = old_obj_typed_ptr;

    /* get the type of the old obj */
    os_class_type &c = old_obj_typed_ptr.get_type();

    /* get the old data member values */
    brep *the_old_boundary_rep_ val;
    os_fetch(
        the_old_obj,
        *c.find_member("boundary_rep"),
        the_old_boundary_rep_val
    );

    /* set the new data member value */
    the_new_obj->cells = the_old_boundary_rep_val;
}
```

### Example application

Now here is an application that associates the reclassifier and transformers with their classes and invokes evolution:

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

#include "part.hh"
#include "new_epart.hh"
#include "new_mpart.hh"

main() {

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

    /*  associate part_reclassifier() with the class part */
    os_schema_evolution::augment_subtype_selectors(
        os_evol_subtype_fun_binding("part", part_reclassifier)
    );

    /*  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/workdb",
        "/example/partsdb"
    );
}
```

# Illegal 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 illegal 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 illegal pointer or reference is encountered.

## Ignoring Illegal Pointers During Schema Evolution

If you want evolution to continue after detection of an illegal pointer or reference, you can specify that illegal pointers be ignored, by calling **os_schema_evolution::set_ignore_illegal_pointers()** with a nonzero argument, before calling **evolve()**. This function is declared as follows:

```
static void os_schema_evolution::set_ignore_illegal_pointers(
    os_boolean);
```

## Using a Handler Function for Illegal Pointers

Alternatively, you can provide a handler function associated with one or more of the following categories of illegal pointers and references:

- Illegal pointers and C++ references to objects
- Illegal ObjectStore local references
- ObjectStore nonlocal references
- Illegal pointers and C++ references to members
- Illegal database root values

Each time an illegal pointer or reference of the associated kind is detected, the handler function is executed on it, and then schema evolution is resumed. A handler function cannot modify any data in the databases being evolved, except for the illegal pointer or reference itself, which can be assigned a new value. The function can, however, generate text output. For example, you can record the location of an illegal pointer by creating a transient ObjectStore reference to the illegal pointer, and then dumping its text representation to a file (see **os_reference::dump()** in the

*ObjectStore C++ API Reference*). This text representation can be used by a subsequent process to create another ObjectStore reference to the same illegal pointer (see **os_reference::os_reference()** in the *ObjectStore C++ API Reference*).

## Creating a Handler Function

To associate a handler function with a category of illegal pointer or reference:

- Define a function with the appropriate signature.
- Register the function with a call to the static member function **os_schema_evolution::set_illegal_pointer_handler()**.

The signatures of the handler functions for each category are as follows:

Illegal pointers and C++ references to objects

```
void function_name(
    objectstore_exception &exc,
    char *msg,
    void *&the_bad_ptr
);
```

Illegal ObjectStore local references

```
void function_name(
    objectstore_exception &exc,
    char *msg,
    os_reference_local &the_bad_ref
);
```

Illegal ObjectStore nonlocal references

```
void function_name(
    objectstore_exception &exc,
    char *msg,
    os_reference &the_bad_ref
);
```

Illegal pointers and C++ references to members

```
void function_name(
    objectstore_exception &exc,
    char *msg,
    os_os_canonical_ptom &the_bad_ptr
);
```

Illegal ObjectStore root values

```
void function_name(
    objectstore_exception &exc,
    char *msg,
    os_database_root &the_bad_root
);
```

## The set_illegal_pointer_handler() Function

The function **os_schema_evolution::set_illegal_pointer_handler()** has four overloadings corresponding to the four categories of illegal pointers and references. Each takes one argument, a pointer to the handler function of the appropriate signature.

Function arguments

For each kind of illegal pointer handler, the **exc** argument is a reference to the exception that would have been signaled had you not provided a handler. The exception is always a child exception of err_se_illegal_pointer. The **msg** argument is the error message that would have been sent to **stderr**. The last argument, **the_bad_ref** or **the_bad_ptr**, is a C++ reference to the illegal pointer or illegal ObjectStore reference.

## Identifying Illegal Pointers Passed to a Handler

To help you identify an illegal pointer passed to a handler function, the class **os_schema_evolution** provides three useful functions not yet introduced:

- **os_schema_evolution::get_path_to_member()**
- **os_schema_evolution::path_name()**
- **os_schema_evolution::get_evolved_address()**

**get_path_to_member()** function

**get_path_to_member()** performed on a **void*** returns an instance of **os_path** representing the data member whose value is pointed to by the **void***.

**path_name()** function

**path_name()** performed on an **os_path** returns a string naming this data member.

**get_evolved_address()** function

**get_evolved_address()**, like **get_evolved_object()**, returns the address of the new version of a specified unmigrated object. **get_evolved_address()** is used here because **get_evolved_object()** signals an exception when performed on an illegal pointer. (**get_unevolved_address()**, like **get_unevolved_object()**, returns the address of the old version of the specified migrated object.)

The **os_schema_evolution** class is described in Chapter 2 of the *ObjectStore C++ API Reference*.

Besides the categorization we have been discussing, there is another, orthogonal way of dividing illegal pointers and

references into categories. This division will help you understand what pointers and references get counted as illegal.

### 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 illegal and can result in 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 360 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 illegal and can result in the exception err_se_ambiguous_void_pointer.

Consider again Example: Changing Inheritance on page 360. 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**. But while before evolution the pointer could be interpreted as referring to the value of **epart::part_id**, after evolution it could no longer be interpreted as referring to this object. Since the value of **epart::part_id** is no longer one of the pointer's associated objects, the pointer becomes illegal. (Remember that ObjectStore makes no semantic connection between **epart::part_id** and **part::part_id**.)

Note that **void*** pointers appear in every database, since the values of database roots are typed as **void***. They might be common in some databases, since in the underlying representations of ObjectStore collections, elements are typed as **void***.

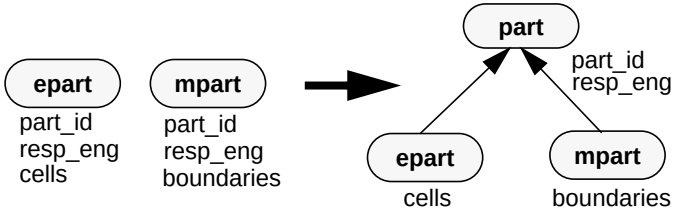
Pointers and references to transient or freed memory and type-mismatched pointers and references: these are pointers and references that are illegal even before schema evolution, but ObjectStore will detect them during instance initialization. Pointers and references to transient objects, or to objects that have been deleted, are illegal. Pointers and references with particular types that are not actually the addresses of some objects of that type are also illegal.

# Example: Using Illegal Pointer Handlers

Consider the schema change made in Example: Changing Inheritance on page 360.

Changing **epart** and **mpart** to inherit from **part**

Changing **epart** and **mpart** to inherit from part; factoring out the common state to the base type.




As described above, if a database undergoes this schema change, and it contains **void*** pointers to **epart**s or **mpart**s, these pointers will be detected as illegal, and should be handled with an illegal pointer handler.

A **void*** pointer to (for example) an **epart** is illegal because it could be interpreted, before evolution, as referring to the value of **epart::part_id**, which does not exist after evolution. But if we know this interpretation is never intended, then we can use the following illegal pointer handler.

Example: using an illegal pointer handler

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

#include <stdio.h>
#include <string.h>

#include "part5new.hh"

static void my_illegal_pointer_handler(
        objectstore_exception& exc,
        char* explanation,
        void*& illegalp
) {
    if (& exc == & err_se_ambiguous_void_pointer)
    {
        os_path * member_path =
            os_schema_evolution::get_path_to_member(illegalp);

        if (member_path)
```

```
        {
            char * path_string = os_schema_evolution::path_name(
                * member_path);
            if (strcmp(path_string, "epart.supplier_id") == 0 ||
                strcmp(path_string, "mpart.supplier_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 */
                illegalp = (void *)
                os_schema_evolution::
                get_evolved_address(illegalp);
                return;
            } /* end if */

        } /* end if */

    } /* end if */

    /* an unanticipated illegal pointer, signal the exception */
    exc.signal(explanation);
}
```

### Using transformers with illegal pointer handlers

For this example, we use the same transformers as Example 3. Below is an application that associates the transformers with their classes, registers the illegal pointer handler, and invokes evolution.

```
#include <ostore/ostore.hh>
#include <ostore/coll.hh>
#include <ostore/schmevol.hh>
#include <ostore/mop.hh>
#include <stdio.h>
#include <string.h>
#include "part5new.hh"

main(int, char * argv[]) {

    /* register the illegal pointer handler */
     os_schema_evolution::set_illegal_pointer_handler(
        my_illegal_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(argv[2], argv[1]);
}
```

# 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. In the case of an obsolete query, ObjectStore internally marks the query so that subsequent attempts to use it result in the exception err_os_query_evaluation_error.

As with illegal pointers, you can handle obsolete queries or indexes by providing a special handler function for each purpose. If you do not supply handlers, ObjectStore signals an exception when it detects an obsolete query or index.

### Handling obsolete queries or indexes

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()**.

### Form of obsolete query handler call

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.

### Form of obsolete index handler call

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, together 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 will be 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 will be initialized to zero

### 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 will be migrated. Definitions for all these functions appear in the task list.

A program to generate a task list is just 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()**

Prior to 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()**, but 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:

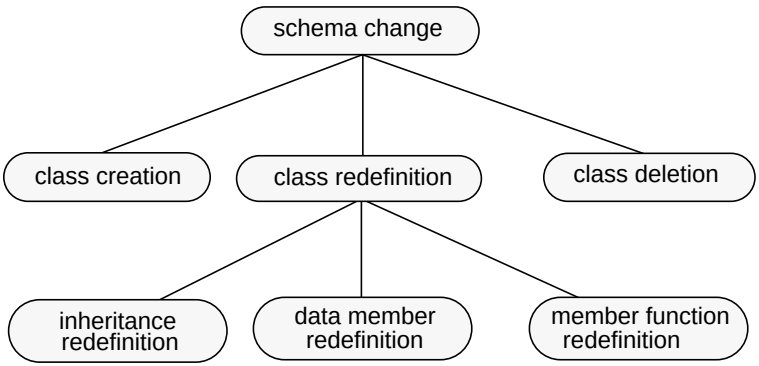
- Class creation
- Class redefinition
- Class deletion

### 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

### 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. Since 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.

## Inheritance Redefinition

But, 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, and changing a class so that it no longer inherits from an existing class, or changing class inheritance from virtual to nonvirtual or the reverse. See Instance Reclassification on page 340.

## 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.) See Instance Reclassification on page 340.

## 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 do not require schema evolution. See Instance Reclassification on page 340.

## 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 result in 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 illegal 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. See Instance Reclassification on page 340.

## Instance Reclassification

As mentioned earlier, the schema evolution facility provides a special capability for *reclassifying* instances of a base class so that they become instances of classes derived from the base class. This form of instance migration is never actually *required* by a schema change, but it is often desirable.

The sections that follow discuss the default initialization rules for each of these categories (except class creation, which, as explained, does not require the use of the evolution facility). See Instance Reclassification on page 340.

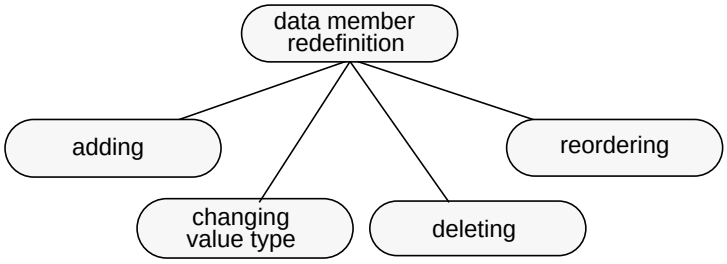
# Schema Changes Related to Data Members

The sections that follow consider the different types of schema modification related to data members:



This section is particularly concerned with describing the instance migration phase of schema evolution for each kind of modification.

## Categories of data member redefinition




Notice 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,** will be 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. How 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. Since no new storage is created by this schema change, the issue of initialization does not arise. Note however that a transformer function for the modified class can still access the value of the removed member in the unevolved instance. See Example: Changing Inheritance on page 360.

By default, pointers to members being removed result in an illegal pointer exception during evolution. You can, however, supply an illegal pointer handler to process the illegal pointer and resume evolution. See Illegal Pointers on page 373.

# 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. How 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 will be 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 will assign 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 (integral 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 do not need to perform schema evolution. This is because such a change does not change the size of the associated field, and does not change how (sufficiently small) positive numbers are represented.

Now consider the case in which the new value type *is* an array type.

Array values with compatible types

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 generate 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, then 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).

Non-array 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, arrays are initialized 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. Since there is no new storage created by this schema change, the issue of initialization does not arise.

# Summary of Data Member Changes *Not* Requiring Explicit Evolution

Note that you do not need to 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 so do not require schema evolution.

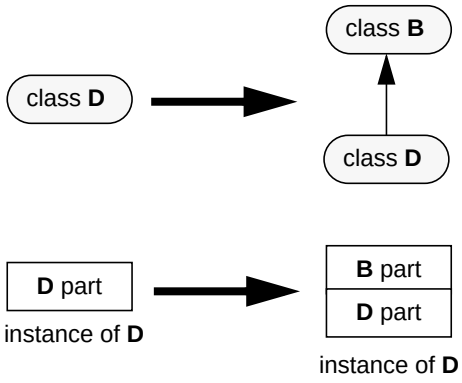
# Schema Changes Related to Class Inheritance

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

When you modify a database's schema by adding a base class, say **B**, to an existing class, say **D**, 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 will add the **B** part to each instance of **D**, and initialize that part as if by a constructor that initializes each member to **0**.

If you provide a transformer function for **D**, it will be run during the instance transformation phase.

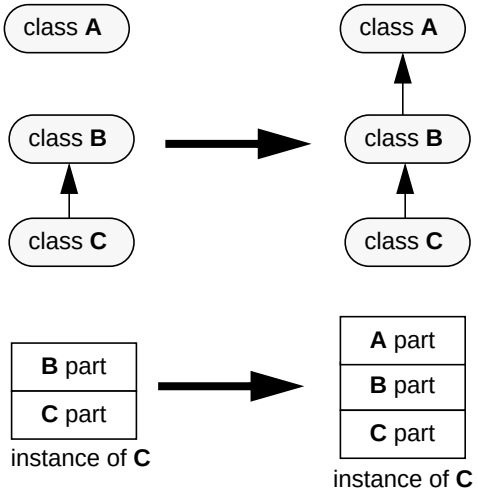
Note that this category of schema change covers more cases than might be suggested by the illustration above.

In particular,

- Schema evolution works just 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 399).
- 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 (but see Instance Reclassification on page 366).

When changing a class requires migrations

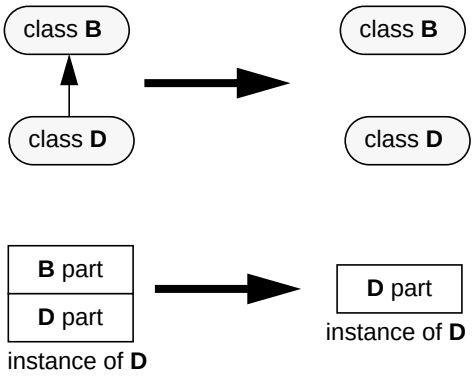



When you change the definition of **B** so that it inherits from **A**, instances of **C** (derived from **B**) must be migrated.

# 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



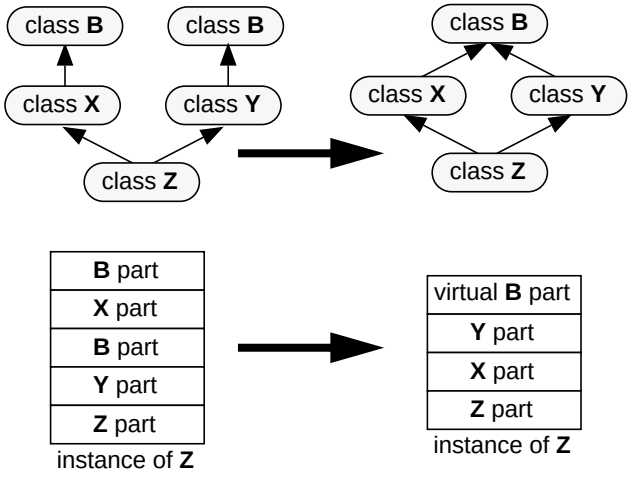

When the class **D** is modified so that it no longer inherits from a base class, B, its instances must be modified to remove the **B** part.

Pointers to the subobject being removed, if they are typed as **B*** rather than **D***, result in an illegal pointer exception's being signaled during evolution. (Pointers typed as **D*** are, of course, automatically adjusted to point to the migrated instance of **D**.) The same is true for pointers (so typed) to data members of the deleted subobject.

# 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** will have 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 below illustrates 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

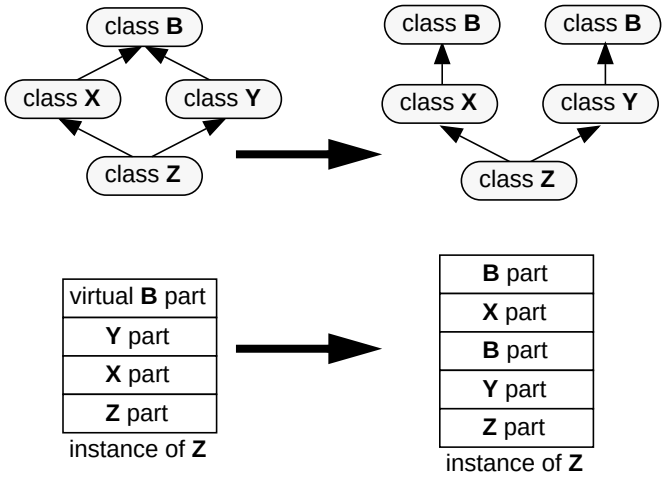



When you change both **X** and **Y** to inherit virtually from **B**, instances of **Z** (derived from both **X** and **Y**) are migrated so that they have only a single **B** part.

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**. So 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

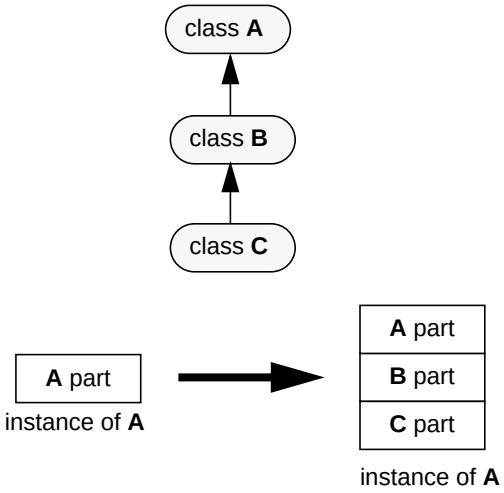
Deletion of a class using the schema evolution facility results in deletion of all its instances during the instance initialization phase. Pointers and references to objects so deleted provoke err_se_deleted_object.

# Instance Reclassification

When an instance is reclassified, it acquires at least one new subobject during instance initialization. Each new subobject is initialized as if by a constructor that initializes each field to **0**.

Note that an instance of a base class can be reclassified as an instance of any class derived from the base class, not just classes *directly* derived from it.

### Effect of instance reclassification




When you reclassify an instance of **A** so that it becomes an instance of **C**, it acquires two additional subobjects.

# Chapter 10
# Database Utility API

The information about the database utility APIs is organized in the following manner:

# Database Utility API Overview

The database utility API provides C++ functions corresponding to the utilities documented in Chapter 4, Utilities, in *ObjectStore Management*. You can use them as a basis for your own database utilities and tools.

## The **os_dbutil** class

All the functions in this facility are members of the class **os_dbutil**. See **os_dbutil** in Chapter 2 of the *ObjectStore C++ API Reference* for a complete description of the **os_dbutil** class.

## Initializing the database utility API

Call the following function before using any other members of **os_dbutil**:

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

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

# Managing Servers

## Getting Rawfs Disk Space Information with disk_free()

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

Function arguments

Gets disk space usage in the rawfs managed by the Server on the machine named **hostname. blocks** points to an instance of **os_free_blocks** allocated by the caller, using the zero-argument constructor. The class **os_free_blocks** has the following public data members:

```
os_unsigned_int32    struct_version;
os_unsigned_int32    free_blocks;
os_unsigned_int32    file_system_size;
os_unsigned_int32    used_blocks;
```

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

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

See also

- **os_dbutil::disk_free()** in Chapter 2 of *ObjectStore C++ API Reference*
- osdf: Displaying Rawfs Disk Space Information in Chapter 4 of *ObjectStore Management*

## Getting Server Information with svr_stat()

```
static void svr_stat(
    const char *server_host,
    os_unsigned_int32 request_bits
    os_svr_stat *svrstat_data
) ;
```

Function arguments

The **db_util::svr_stat()** function gets statistics for a Server's clients on **server_host**.

See also

- **os_dbutil::svr_stat()** in Chapter 2 of *ObjectStore C++ API Reference*
- ossvrstat: Displaying Server and Client Information in Chapter 4 of *ObjectStore Management*
- ossvrmtr: Displaying Server Resource Information in Chapter 4 of *ObjectStore Management*

Enumerators for **request_bits**

The **request_bits** argument specifies what information is desired. Supply this argument by forming the bit-wise disjunction of zero or more of the following enumerators:

- **os_svr_stat::get_svr_usage**
- **os_svr_stat::get_svr_meter_samples**
- **os_svr_stat::get_svr_parameters**
- **os_svr_stat::get_client_info_self**
- **os_svr_stat::get_client_info_others**

For each enumerator that is specified, the corresponding information is retrieved.

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

Public data members in **os_svr_stat**

The **svr_stat_data** argument points to an instance of **os_svr_stat** allocated by the caller, using the zero-argument constructor. This structure has the following public data members:**os_dbutil::os_**

```
os_unsigned_int32          struct_version;
os_svr_stat_svr_header     header;
os_svr_stat_svr_parameters* svr_parameters;
os_svr_stat_svr_rusage*    svr_rusage;
os_svr_stat_svr_meters*    svr_meter_samples;
os_unsigned_int32          n_meter_samples;
os_svr_stat_client_info*   client_info_self;
os_svr_stat_client_info*   client_info_others;
os_unsigned_int32          n_clients;
```

**os_db_util::os_svr_stat()** allocates instances of these as required.

They are automatically cleaned up when the **os_svr_stat** instance (as provided by the caller) is deleted.

### Public data members in **os_svr_stat_svr_header**

The **os_svr_stat_svr_header** data member (a member of **os_svr_stat**) has the following public data members:

```
os_unsigned_int32       struct_version;
char*                   os_release_name;
os_unsigned_int32       server_major_version;
os_unsigned_int32       server_minor_version;
char*                   compilation;
```

### Public data members in **os_svr_stat_svr_parameters**

The **os_svr_stat_svr_parameters** data member (a member of **os_svr_stat**) has the following public data members:

```
os_unsigned_int32       struct_version;
char*                   parameter_file;
os_boolean              allow_shared_mem_usage;
os_int32*               authentication_list;
os_unsigned_int32       n_authentications;
os_int32                RAWFS_db_expiration_time;
os_int32                deadlock_strategy;
os_unsigned_int32       direct_to_segment_threshold;
char*                   log_path;
os_unsigned_int32       current_log_size_sectors;
os_unsigned_int32       initial_log_data_sectors;
os_unsigned_int32       growth_log_data_sectors;
os_unsigned_int32       log_buffer_sectors;
os_unsigned_int32       initial_log_record_sectors;
os_unsigned_int32       growth_log_record_sectors;
os_unsigned_int32       max_data_propagation_threshold;
os_unsigned_int32       max_propagation_sectors;
os_unsigned_int32       max_msg_buffer_sectors;
os_unsigned_int32       max_msg_buffers;
os_unsigned_int32       sleep_time_between_2p_
                        outcomes;
os_unsigned_int32       sleep_time_between_propagates;
os_unsigned_int32       write_buffer_sectors;
os_unsigned_int32       tcp_recv_buffer_size;
```

```
os_unsigned_int32       tcp_send_buffer_size;
os_boolean              allow_nfs_locks;
os_boolean              allow_remote_database_access;
os_unsigned_int32       max_two_phase_delay;
os_unsigned_int32       max_aio_threads;
os_unsigned_int32       cache_mgr_ping_time;
os_unsigned_int32       max_memory_usage;
os_unsigned_int32       max_connect_memory_usage;
os_unsigned_int32       remote_db_grow_reserve;
os_boolean              allow_estale_to_corrupt-DBs;
os_int32                restricted_file_db_access_only;
os_unsigned_int32       failover_heartbeat_time;
```

### Public data members in **os_svr_stat_svr_rusage**

The **os_svr_stat_svr_rusage** data member (a member of **os_svr_stat**) has the following public data members:

```
os_unsigned_int32       struct_version;
os_timesecs             ru_utime; /* user time used */
os_timesecs             ru_stime; /* system time used */
os_int32                ru_maxrss; /* max resident set size */
os_int32                ru_ixrss; /* shared mem size */
os_int32                ru_idrss; /* unshared data size */
os_int32                ru_isrss; /* unshared stack size */
os_int32                ru_minflt; /* page reclaims */
os_int32                ru_majflt; /* page faults */
os_int32                ru_nswap; /* swaps */
os_int32                ru_inblock; /* block input ops */
os_int32                ru_msgsnd; /* messages sent */
os_int32                ru_msgrcv; /* messages received */
os_int32                ru_nsignals; /* signals received */
os_int32                ru_nvcsw;
                        /* voluntary context switches */
os_int32                ru_nivcsw;
                        /* involuntary context switches */
```

### Public data members in **os_svr_stat_svr_meters**

The **os_svr_stat_svr_meters** data member (a member of **os_svr_stat**) has the following public data members:

```
os_unsigned_int32       struct_version;
```

```
os_boolean          valid;
os_unsigned_int32   n_minutes;
os_unsigned_int32   n_receive_msgs;
os_unsigned_int32   n_callback_msgs;
os_unsigned_int32   n_callback_sectors_requested;
os_unsigned_int32   n_callback_sectors_succeeded;
os_unsigned_int32   n_sectors_read;
os_unsigned_int32   n_sectors_written;
os_unsigned_int32   n_commit;
os_unsigned_int32   n_phase_2_commit;
os_unsigned_int32   n_readonly_commit;
os_unsigned_int32   n_abort;
os_unsigned_int32   n_phase_2_abort;
os_unsigned_int32   n_deadlocks;
os_unsigned_int32   n_readonly_commit;
os_unsigned_int32   n_abort;
os_unsigned_int32   n_phase_2_abort;
os_unsigned_int32   n_deadlocks
os_unsigned_int32   n_msg_buffer_waits;
os_unsigned_int32   n_log_records;
os_unsigned_int32   n_log_seg_switches;
os_unsigned_int32   n_flush_log_data_writes;
os_unsigned_int32   n_flush_log_record_writes;
os_unsigned_int32   n_log_data_writes;
os_unsigned_int32   n_log_record_writes;
os_unsigned_int32   n_sectors_propagated;
os_unsigned_int32   n_sectors_direct;
os_unsigned_int32   n_do_some_propagation;
```

### Public data members in **os_svr_stat_client_info**

The **os_svr_stat_client_info** data member (a member of **os_svr_stat**) has the following public data members:

```
os_unsigned_int32               struct_version;
os_svr_stat_client_process*     process;
os_svr_stat_client_state*       state;
os_svr_stat_client_meters*      meters;
```

**os_db_util::os_svr_stat()** allocates instances of these as required. They are automatically cleaned up when the **os_svr_stat** instance (as provided by the caller) is deleted.

Public data members in **os_svr_stat_client_process**

The **os_svr_stat_client_process** data member (a member of **os_svr_stat_client_info**) has the following public data members:

```
os_unsigned_int32          struct_version;
char*                      host_name;
os_unsigned_int32          process_id;
char*                      client_name;
os_unsigned_int32          client_id;
```

Public data members in **os_svr_stat_client_state**

The **os_svr_stat_client_state** data member (a member of **os_svr_stat_client_info**) has the following public data members:

```
os_unsigned_int32          struct_version;
os_client_state_type       client_state;
char*                      message_name;
os_boolean                 txn_in_progress;
os_unsigned_int32          txn_priority;
os_unsigned_int32          txn_duration;
os_unsigned_int32          txn_work;
os_client_lock_type        lock_state;
os_unsigned_int32          db_id;
char*                      db_pathname;
os_unsigned_int32          locked_seg_id;
os_unsigned_int32          locking_start_sector;
os_unsigned_int32          locking_for_n_sectors;
os_unsigned_int32          n_conflicts;
os_svr_stat_client_
process*                   lock_conflicts;
```

**os_client_lock_type**

```
enum os_client_lock_type {
    OSSVRSTAT_CLIENT_LOCK_TO_MAX_BLOCKS,
    OSSVRSTAT_CLIENT_LOCK_TO_NBLOCKS,
};
```

**os_client_state_type**

```
enum os_client_state_type {
    OSSVRSTAT_CLIENT_WAITING_MESSAGE,
    OSSVRSTAT_CLIENT_EXECUTING_MESSAGE,
    OSSVRSTAT_CLIENT_WAITING_RANGE_READ_LOCK,
    OSSVRSTAT_CLIENT_WAITING_RANGE_WRITE_LOCK,
```

```
    OSSVRSTAT_CLIENT_WAITING_SEGMENT_WRITE_LOCK,
    OSSVRSTAT_CLIENT_DEAD,
    OSSVRSTAT_CLIENT_WAITING_SEGMENT_READ_LOCK,
    OSSVRSTAT_CLIENT_WAITING_DB_READ_LOCK,
    OSSVRSTAT_CLIENT_WAITING_DB_WRITE_LOCK,
};
```

Data in **locking_start_sector** and **locking_for_n_sectors** is valid only when **lock_state** is **OSSVRSTAT_CLIENT_STATE_WAITING_RANGE_READ_LOCK** or **OSSVRSTAT_CLIENT_STATE_WAITING_RANGE_WRITE_LOCK**.

Public data members in **os_svr_stat_client_meters**

The **os_svr_stat_client_meters** data member (a member of **os_svr_stat_client_info**) has the following public data members:

```
os_unsigned_int32    struct_version;
os_unsigned_int32    n_receive_msgs;
os_unsigned_int32    n_callback_msgs;
os_unsigned_int32    n_callback_sectors_requested;
os_unsigned_int32    n_callback_sectors_succeeded;
os_unsigned_int32    n_sectors_read;
os_unsigned_int32    n_sectors_written;
os_unsigned_int32    n_deadlocks;
os_unsigned_int32    n_lock_timeouts;
os_unsigned_int32    n_commit;
os_unsigned_int32    n_phase_2_commit;
os_unsigned_int32    n_readonly_commit;
os_unsigned_int32    n_abort;
os_unsigned_int32    n_phase_2_abort;
```

## Determining Sector Size with get_sector_size()

```
static os_unsigned_int32 get_sector_size() ;
```

Return values

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

See also

- **os_dbutil::get_sector_size()** in Chapter 2 of *ObjectStore C++ API Reference*
- Chapter 2, Server Parameters, of *ObjectStore Management*

## Killing a Client Thread on a Server with svr_client_kill()

```
static os_boolean svr_client_kill(
    const char *server_host,
    os_int32 client_pid,
    const char *client_name,
    const char *client_hostname,
    os_boolean all,
    os_int32 &status
) ;
```

Function arguments

Kills one or all clients of the specified Server. **server_host** is the name of the machine running the Server.

**client_pid** is the process ID of the client to kill. **client_name** is the name of the client to kill. **client_hostname** is the name of the machine running the client to kill.

If **all** is **1**, the other arguments except **server_host** are ignored, and all clients on the specified Server are killed.

**status** is set to **–2** if a client was killed, **0** if no clients matched the specifications, **2** if multiple clients matched the specification. Any other value means that access was denied.

Return values

Returns **0** for failure and nonzero for success.

See also

- **os_dbutil::svr_client_kill()** in Chapter 2 of *ObjectStore C++ API Reference*
- ossvrclntkill: Disconnecting a Client Thread on a Server in Chapter 4 of *ObjectStore Management*

## Determining Whether a Server Is Running with svr_ping()

```
static char *svr_ping(
    const char *server_host,
    os_svr_ping_state &state
) ;
```

Function arguments

Determines whether an ObjectStore Server is running on the machine named **server_host**. The referent of **state** is set to one of the following global enumerators:

- **os_svr_ping_is_alive**
- **os_svr_ping_not_reachable**
- **os_svr_ping_no_such_host**

Return values

Returns a pointer to the status message string used by the utility **ossvrping**.

See also

- **os_dbutil::svr_ping()** in Chapter 2 of *ObjectStore C++ API Reference*
- ossvrping: Determining If a Server Is Running in Chapter 4 of *ObjectStore Management*

## Shutting Down the Server with svr_shutdown()

```
static os_boolean svr_shutdown(
    const char *server_host
) ;
```

Return values

Shuts down the Server on the machine named **server_host**. Returns nonzero for success, **0** otherwise. On some operating systems, you must have special privileges to use this function.

See also

- **os_dbutil::svr_shutdown()** in Chapter 2 of *ObjectStore C++ API Reference*
- ossvrshtd: Shutting Down the Server in Chapter 4 of *ObjectStore Management*

## Moving Data Out of the Server Transaction Log with svr_checkpoint()

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

Return values

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

See also

- **os_dbutil::svr_checkpoint()** in Chapter 2 of *ObjectStore C++ API Reference*
- ossvrchkpt: Moving Data Out of the Server Transaction Log in Chapter 4 of *ObjectStore Management*

# Managing Clients

## Setting a Client Name with set_client_name()

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

Sets the client name string for message printing.

See also

- **os_dbutil::set_client_name()** in Chapter 2 of *ObjectStore C++ API Reference*

## Getting a Client Name with get_client_name()

**static char const* get_client_name();**

Return values

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

See also

- **os_dbutil::get_client_name()** in Chapter 2 of *ObjectStore C++ API Reference*

## Closing a Server Connection with close_server_connection()

**static void close_server_connection(const char *hostname);**

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

See also

- **os_dbutil::close_server_connection()** in Chapter 2 of *ObjectStore C++ API Reference*

## Closing All Server Connections with close_all_server_connections()

**static void close_all_server_connections();**

Closes all connections the application has to ObjectStore Servers.

See also

- **os_dbutil::close_all_server_connections()** in Chapter 2 of *ObjectStore C++ API Reference*

# Managing Cache Managers

## Getting Cache Manager Status with cmgr_stat()

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

Function arguments

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

See also

- **os_dbutil::cmgr_stat()** in Chapter 2 of *ObjectStore C++ API Reference*
- oscmstat: Displaying Cache Manager Status in Chapter 4 of *ObjectStore Management*

Public data members in **os_cmgr_stat**

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

| | |
|---|---|
| **os_unsigned_int32** | **struct_version;** |
| **os_unsigned_int32** | **major_version;** |
| **os_unsigned_int32** | **minor_version;** |
| **os_unsigned_int32** | **pid;** |
| **char*** | **executable_name** |
| **char*** | **host_name;** |
| **os_unixtime_t** | **start_time;** |
| **os_int32** | **soft_limit;** |
| **os_int32** | **hard_limit;** |
| **os_int32** | **free_allocated;** |
| **os_int32** | **used_allocated;** |
| **os_int32** | **n_clients;** |
| **os_cmgr_stat_client*** | **per_client; /* array */** |
| **os_int32** | **n_servers;** |
| **os_cmgr_stat_svr*** | **per_server; /* array */** |
| **os_int32** | **n_cache_file_usage;** |
| **os_cmgr_stat_file_usage*** | **cache_file_usage; /* array */** |

```
os_int32                    n_commseg_file_usage;
os_cmgr_stat_file_
usage*                      commseg_file_usage; /* array */
char*                       cback_queue;
char*                       extra;
```

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

Public data members in **os_cmgr_stat_client**

The **os_cmgr_stat_client** data member (a member of **os_cmgr_stat**) has the following public data members:

```
os_unsigned_int32           struct_version;
os_int32                    pid;
os_unsigned_int32           euid;
char*                       name;
os_int32                    major_version;
os_int32                    minor_version;
os_int32                    commseg;
```

Public data members in **os_cmgr_stat_svr**

The **os_cmgr_stat_svr** data member (a member of **os_cmgr_stat**) has the following public data members:

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

Public data members in **os_cmgr_stat_file_usage**

The **os_cmgr_stat_file_usage** data member (a member of **os_cmgr_stat**) has the following public data members:

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

## Deleting Unused Cache and commseg Files with cmgr_remove_file()

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

Function arguments

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

Return values

This function returns a pointer to the result message string.

See also

- **os_dbutil::cmgr_remove_file()** in Chapter 2 of *ObjectStore C++ API Reference*
- oscmrf: Deleting Cache and Commseg Files in Chapter 4 of *ObjectStore Management*

## Shutting Down the Cache Manager with cmgr_shutdown()

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

Function arguments

Shuts down the Cache Manager running on the machine with the specified **hostname**. The argument **cm_version_number** must match the Cache Manager's version number.

Return values

Returns a pointer to the result message string.

See also

- **os_dbutil::cmgr_shutdown()** in Chapter 2 of *ObjectStore C++ API Reference*
- oscmshtd: Shutting Down the Cache Manager in Chapter 4 of *ObjectStore Management*

# Managing Databases

## Changing Database Group Names with chgrp()

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

Changes the primary group of the rawfs directory or database whose name is **pathname**. **gname** is the name of the new group.

See also

- **os_dbutil::chgrp()** in Chapter 2 of *ObjectStore C++ API Reference*
- oschgrp: Changing Database Group Names in Chapter 4 of *ObjectStore Management*

## Changing Database Owner with chown()

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

Function arguments

Changes the owner of the rawfs directory or database whose name is **pathname**. **uname** is the user name of the new owner.

See also

- **os_dbutil::chown()** in Chapter 2 of *ObjectStore C++ API Reference*
- oschown: Changing Database Owners in Chapter 4 of *ObjectStore Management*

## Changing Database Permissions with chmod()

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

Function arguments

Changes the protections on the rawfs database or directory whose name **pathname**. **mode** specifies the new protections.

See also

- **os_dbutil::chmod()** in Chapter 2 of *ObjectStore C++ API Reference*
- oschmod: Changing Database Permissions in Chapter 4 of *ObjectStore Management*

## Changing a Rawfs Hosts with rehost_link()

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

Function arguments

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

See also

- **os_dbutil::rehost_link()** in Chapter 2 of *ObjectStore C++ API Reference*

## Changing All Rawfs Hosts with rehost_all_links()

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

Function arguments

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

See also

- **os_dbutil::rehost_all_links()** in Chapter 2 of *ObjectStore C++ API Reference*
- oschhost: Changing Rawfs Link Hosts in Chapter 4 of *ObjectStore Management*

## Copying Databases with copy_database()

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

Function arguments

Copies the database named **src_database_name** and names the copy **dest_database_name**. If there is already a database named **dest_database_name**, it is silently overwritten.

Return values

Returns **0** for success, **1** if per-segment access control information has been changed during copy (which can happen when copying a rawfs database to a file database, since file databases do not have separate segment-level protections). This function can be called either within or outside a transaction.

See also

- **os_dbutil::copy_database()** in Chapter 2 of *ObjectStore C++ API Reference*
- oscp: Copying Databases in Chapter 4 of *ObjectStore Management*

## Expanding File Names with expand_global()

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

Function arguments and return values

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

See also

- **os_dbutil::expand_global()** in Chapter 2 of *ObjectStore C++ API Reference*
- osglob: Expanding File Names in Chapter 4 of *ObjectStore Management*

## Creating Rawfs Directories with mkdir()

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

Function arguments

Makes a rawfs directory whose pathname is **path**. The new directory's protections is specified by mode. If **create_missing_dirs** is nonzero, creates the missing intermediate directories.

Return values

Signals err_directory_not_found if **create_missing_dirs** is **0** and there are missing intermediate directories. Signals err_directory_exists if there is already a directory with the specified path. Signals err_database_exists if there is already a database with the specified path.

See also

- **os_dbutil::mkdir()** in Chapter 2 of *ObjectStore C++ API Reference*
- osmkdir: Creating a Rawfs Directory in Chapter 4 of *ObjectStore Management*

## Setting Links in the Rawfs with make_link()

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

Function arguments

Makes a rawfs soft link. **target_name** is the path pointed to by the link. **link_name** is the pathname of the link.

Return values

Signals err_database_exists or err_directory_exists if **link_name** already points to an existing database or directory.

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

To access a particular database or directory, a client can follow as many as 15 cross-Server links. For example, a client traverses a link to Server **Q**. Server **Q** sends the client to Server **P**. Server **P** sends the client to another Server or even back to Server **Q**. Each connection to a Server counts as one link. It does not matter whether or not the Server was previously connected to in the link chain. When the client reaches the sixteenth link, ObjectStore displays the error message err_too_many_cross_svr_links.

To access a particular database or directory in its rawfs, the Server can traverse as many as 10 (same-Server) links. When the Server reaches the eleventh link, ObjectStore displays the error message err_too_many_links.

In a chain of links, a client can return to a Server that it contacted earlier in the chain. In this situation, the Server's count of links within its rawfs begins with one. It does not continue the count from where it left off during the previous connection. Each time a link sends the client to a Server, the Server can follow as many as ten links within its rawfs.

These limits allow ObjectStore to catch circular links. For example, **A** is a link to **B**, and **B** is either directly or indirectly a link to **A**.

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

See also

- **os_dbutil::make_link()** in Chapter 2 of *ObjectStore C++ API Reference*
- osln: Creating Links in the Rawfs in Chapter 4 of *ObjectStore Management*
- **rehost_link** and **rehost_all_links** in oschhost: Changing Rawfs Link Hosts in Chapter 4 of *ObjectStore Management*
- osln: Creating Links in the Rawfs in Chapter 4 of *ObjectStore Management*

## Removing Databases and Rawfs Links with remove()

**static void remove(char const *path) ;**

Return values

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

See also

- **os_dbutil::remove()** in Chapter 2 of *ObjectStore C++ API Reference*
- osrm: Removing Databases and Rawfs Links in Chapter 4 of *ObjectStore Management*

## Removing Rawfs Directories with rmdir()

**static void rmdir(const char *path) ;**

Return values

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

See also

- **os_dbutil::rmdir()** in Chapter 2 of *ObjectStore C++ API Reference*
- osrmdir: Removing a Rawfs Directory in Chapter 4 of *ObjectStore Management*

## Moving Directories and Databases with rename()

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

Function arguments

Renames the rawfs database or directory. **source** is the old name and **target** is the new name.

Return values

Signals err_cross_server_rename if **source** and **target** are on different Servers. Signals err_invalid_rename if the operation makes a directory its own descendent. Signals err_database_exists if a database named **target** already exists. Signals err_directory_exists if a directory named **target** already exists.

See also

- **os_dbutil::rename()** in Chapter 2 of *ObjectStore C++ API Reference*
- osmv: Moving Directories and Databases in Chapter 4 of *ObjectStore Management*

## Testing a Pathname for Specified Conditions with stat()

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

Function arguments

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

Return values

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

See also

- **os_dbutil::stat()** in Chapter 2 of *ObjectStore C++ API Reference*
- ostest: Testing a Pathname for Specified Conditions in Chapter 4 of *ObjectStore Management*
- osls: Displaying Directory Content in Chapter 4 of *ObjectStore Management*

## Listing Directory Contents with list_directory()

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

Function arguments

Lists the contents of the rawfs directory named **path**.

Return values

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

See also

- **os_dbutil::list_directory()** in Chapter 2 of *ObjectStore C++ API Reference*
- osls: Displaying Directory Content in Chapter 4 of *ObjectStore Management*

## Find Database Size with ossize()

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

Function arguments

Prints to standard output the size of the database whose name is **pathname**. **options** points to an instance of **os_size_options** allocated by the caller using the zero-argument constructor. **os_size_options** has the following public data members (each member corresponds to an option for the utility **ossize**):

```
os_unsigned_int32   struct_version;
os_boolean          flag_all;              /* -a */
os_boolean          flag_segments;         /* -c */
os_boolean          flag_total_database;   /* -C */
os_boolean          flag_free_block_map;   /* -f */
os_unsigned_int32   one_segment_number;    /* -n */
os_boolean          flag_every_object;     /* -o */
char                flag_summary_order;    /* -s  's'=space 'n'=number
                                              't'=typename */
os_boolean          flag_upgrade_rw;       /* -u */
const char*         workspace_name;        /* -w */
os_boolean          flag_list_workspaces;  /* -W */
os_boolean          flag_internal_segments; /* -0 */
os_boolean          flag_access_control;   /* -A */
```

The constructor sets **struct_version** to the value of **os_size_options_version** in the **dbutil.hh** file included by your application.

If this version is different from that used by the library, err_misc is signaled. The constructor initializes all other members to **0**.

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

If **OS__DBUTIL_NO_MVCC** is set, this function opens the database for read only, rather than for multiversion concurrency control (the default).

See also

- **os_dbutil::ossize()** in Chapter 2 of *ObjectStore C++ API Reference*
- ossize: Displaying Database Size in Chapter 4 of *ObjectStore Management*

## Verifying Pointers and References with osverifydb()

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

Function arguments

Verifies that all pointers and references in the database named **dbname**. **opt** point to an instance of **os_verify_db_options** allocated by the caller using the zero-argument constructor. **os_verify_db_options** has the following public data members:

```
os_boolean verify_segment_zero ;
    /* verify the schema segment */

os_boolean verify_collections ;
    /* check all top-level collections */

os_boolean verify_pointer_verbose;
    /* print pointers as they are verified */

os_boolean verify_object_verbose ;
    /* print objects as they are verified */

os_boolean verify_references ;
    /* check all OS references */

os_int32  segment_error_limit ;
    /* maximum errors per segment */

os_boolean print_tag_on_errors ;
    /* print out the tag value on error */

const char* explicit_workspace
    /* The workspace name if one was explicitly specified. */

const void* track_object_ptr ;
    /* Track object identified by pointer */
```

```
const char* track_object_ref_string;
    /* Track the object identified by the reference string. */

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

You must have called **os_collection::initialize()** and **os_mop::initialize()** prior to calling this function.

If **OS__DBUTIL_NO_MVCC** is set, this function opens the database for read only, rather than multiversion concurrency control (the default).

Return values

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

See also

- **os_dbutil::osverifydb()** in Chapter 2 of *ObjectStore C++ API Reference*
- osverifydb: Verifying Pointers and References in a Database in Chapter 4 of *ObjectStore Management*

# Managing Schemas

## Comparing Schemas with compare_schemas()

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

Function arguments and return values

Compares the schemas of **db1** and **db2**. Returns **1** if the schemas are incompatible, **0** otherwise. Each database can contain an application schema, a compilation schema, or a database schema. If the database contains a database schema, it can be local or remote.

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

See also

- **os_dbutil::compare_schemas()** in Chapter 2 of *ObjectStore C++ API Reference*
- osscheq: Comparing Schemas in Chapter 4 of *ObjectStore Management*

## Setting the Application Schema with set_application_schema_path()

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

Function arguments

Finds or sets an executable's application schema database. **executable_pathname** specifies the executable. **database_pathname** is either the new schema's pathname or **0**.

Return values

If **database_pathname** is **0**, the function returns new storage containing the current pathname. If **database_pathname** is nonzero, the function returns **0**.

See also

- **os_dbutil::set_application_schema_path()** in Chapter 2 of *ObjectStore C++ API Reference*
- ossetasp: Patching Executable with Application Schema Pathname in Chapter 4 of *ObjectStore Management*

# Exceptions Summary

err_file_pathname: A rawfs pathname was expected.

err_misc: An unexpected **0** argument or an invalid pathname was passed; version mismatch of an in / out structure argument.

err_rpc: RPC error.

err_too_many_cross_svr_links: Excessively long cross-server link chain. The maximum depth of a cross-server link chain is currently 15.

err_no_rawfs: There is no database file system (rawfs) on this Server.

err_read_only_file_system: The file database is stored in a read-only file system.

err_no_credentials: Access is not permitted; no credentials were presented.

err_server_not_superuser: The Server is not running as the superuser.

err_link_not_found: Intrarawfs link was not found.

err_too_many_links: Too many levels of intrarawfs links.

err_rawfs_not_upgraded: The rawfs is from an old release.

err_permission_denied: Permission to access this database was denied.

err_invalid_pathname: The rawfs pathname is not valid.

err_directory_not_found: The directory was not found.

err_database_not_found: The rawfs database was not found.

err_link_not_found: The rawfs link was not found.

# Index

## D



## E

# F



# G



# H



# I

## L



## M

## N



## O

## P

# Q



# R

# S

## T

## U



## V