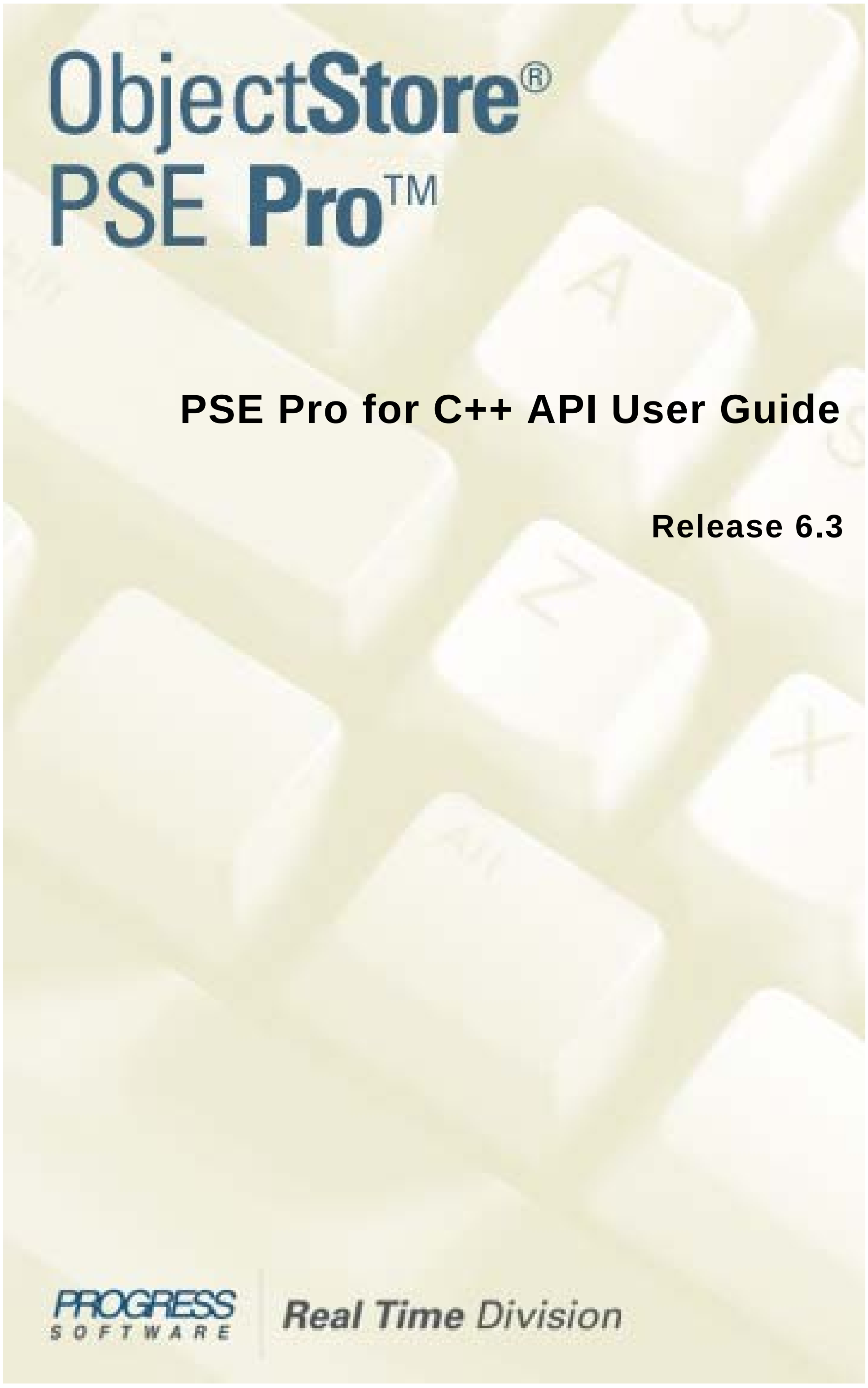


ObjectStore® PSE Pro™

PSE Pro for C++ API User Guide

Release 6.3

PROGRESS SOFTWARE | Real Time Division

Contents

Preface

Purpose

PSE Pro for C++ API User Guide provides information and instructions for using the PSE Pro for C++ API to write database applications.

Audience

This book is for programmers responsible for writing database applications that use ObjectStore PSE Pro for C++. No previous knowledge of PSE Pro for C++ or the C++ interface to ObjectStore is required. It is assumed that you are experienced with Visual C++.

Scope

This book supports Release 6.3 of ObjectStore PSE Pro for C++.

How This Book Is Organized

Chapter 1, Introducing PSE Pro, on page 9, describes the main features in PSE Pro and ObjectStore. It also describes the product's architecture.

Chapter 2, An Example Using PSE Pro, on page 15, provides a simple example that shows how to use PSE Pro.

Chapter 3, Managing Databases, on page 25, includes information about creating, opening, closing, destroying, copying, and dumping databases.

Chapter 4, Storing and Updating Objects in a Database, on page 45, describes how to store, retrieve, and update persistent objects.

Chapter 5, Memory and Database Management Tasks, on page 61, discusses fault handlers, hook functions, environment variables, using PSE Pro and ObjectStore together, obtaining product information, and using typedefs.

Chapter 6, Transactions, on page 79, provides instructions for accessing persistent data inside transactions.

Chapter 7, The Undo/Redo Facility, on page 85, shows how to mark sections of program activity, undo changes in marked sections, and redo changes that were undone.

Chapter 8, Object Iteration, on page 89, discusses how to use an object cursor to iterate over the objects in a database.

Chapter 9, Evolving Database Schemas, on page 93, provides instructions for modifying the classes already stored in a database.

Notation Conventions

This document uses the following conventions:

Convention	*Meaning*
`Courier`	`Courier font` indicates code, syntax, file names, API names, system output, and the like.
`Bold Courier`	**`Bold Courier font`** is used to emphasize particular code, such as user input.
`Italic Courier`	*`Italic Courier font`* indicates the name of an argument or variable for which you must supply a value.
Sans serif	Sans serif typeface indicates the names of user interface elements such as dialog boxes, buttons, and fields.
Italic serif	In text, *italic serif typeface* indicates the first use of an important term.
`[ ]`	Brackets enclose optional arguments.
`{ }*` or `{ }+`	When braces are followed by an asterisk (`*`), the items enclosed by the braces can be repeated `0` or more times; if followed by a plus sign (`+`), one or more times.
`{ 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 can indicate either that material not relevant to the example has been omitted or that the previous item can be repeated.

Progress Software Real Time Division on the World Wide Web

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

Technical Support

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

- You can send questions to `realtime-support@progress.com`. Remember to include your serial number in the subject of the electronic mail message.
- You can call the Technical Support organization to get help resolving problems. If you are in North America, call 1-781-280-4005. If you are outside North America, refer to the Technical Support Web site at `www.progress.com/realtime/techsupport/contact`.
- You can file a report or question with Technical Support by going to `www.progress.com/realtime/techsupport/techsupport_direct`.
- You can access the Technical Support Web site, which includes

- A template for submitting a support request. This helps you provide the necessary details, which speeds response time.
- Solution Knowledge Base that you can browse and query.
- Online documentation for all products.
- White papers and short articles about using Real Time Division products.
- Sample code and examples.
- The latest versions of products, service packs, and publicly available patches that you can download.
- Access to a support matrix that lists platform configurations supported by this release.
- Support policies.
- Local phone numbers and hours when support personnel can be reached.

Education Services

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

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

Searchable Documents

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

Your Comments

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

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

Chapter 1 Introducing PSE Pro

PSE Pro for C++ provides an application programming interface (API) that allows you to store persistent C++ objects. ObjectStore provides a full-featured DBMS. This client/server product, ObjectStore, includes C++, Java, and ActiveX interfaces, as well as a powerful database engine ideal for developing and deploying distributed processing, multiuser access applications. ObjectStore support for C++ includes these products:

- ObjectStore PSE Pro
- C++ interface to ObjectStore Development Client

PSE Pro, like the full ObjectStore, provides persistent storage for C++ objects, but is primarily designed for the development of single-user applications.

To introduce PSE Pro, this chapter discusses the following topics:

What Is PSE Pro?

PSE Pro is a version of ObjectStore for C++ specifically designed for developing single-user applications. Persistent data is available to programmers in such a way that it appears as familiar, normal C++ objects. Persistent C++ objects and regular C++ objects are manipulated in the same way and behave in the same way.

PSE Pro Features

PSE Pro features include

- Multidatabase access
- Transactions and recovery capabilities
- Undo/redo support
- Virtual memory management capabilities
- Schema evolution support
- Collections

PSE Pro uses less than 500 KB of disk space, and provides the following:

- Creation and manipulation of databases as operating system files or streams within OLE compound files
- Allocation of objects in database memory
- Transparent retrieval and storage of persistent objects
- Recoverable save points
- Database-level locking

Support on HP–UX, Linux, Solaris, and Windows platforms include compatibility with the following operating systems and compilers and libraries:

- HP-UX 11.11 32-bit, aC++ 3.56
- HP-UX 11.11, 64-bit, aC++ 3.56
- Linux 2.4.21-4smp, 32-bit, GCC 3.2.3 (RedHat 8.0 i386)
- Linux 2.4.21-4smp, 64-bit, GCC 3.2.3 (RedHat 8.0 i386)
- Solaris, 32-bit, Sun One Studio 8
- Microsoft Windows XP and 2000, Visual C++ 7.1 (unmanaged)
- Microsoft Foundation Class Library (MFC)
- Microsoft Standard Template Library (STL)

Using STL with PSE Pro

If you want to use STL in your application, ObjectStore Technical Support recommends the implementation that comes with MSVC. See the `stl42` example and its accompanying `readme` file for details.

Using MFC with PSE Pro

You can use MFC to build your PSE Pro application. See Using MFC and/or Using MSDEV IDE. in *Building Applications with PSE Pro for C++* for detailed information.

ObjectStore Features

Beyond PSE Pro, ObjectStore offers the full-featured ObjectStore DBMS that extends the PSE Pro API to provide the following features:

- Multiuser access
- Distributed processing
- Data heterogeneity

- High availability
- Large data set support
- Concurrency control
- Query processing
- Integrity control
- Access control
- Data compaction
- Archive logging
- Distributed lock management

You can use PSE Pro and ObjectStore in the same application to migrate your data to an ObjectStore database. Since the APIs are compatible, you can scale your PSE Pro applications from single-user desktop applications to distributed, highly available, multiuser applications without having to rearchitect them.

How to Use PSE Pro

When you use the PSE Pro API, object storage is almost completely automatic. These are the basic steps you follow to store objects in a database:

1 When you create an object, allocate it in a database with the PSE Pro overloading of `::operator new()` instead of with the usual C++ `new()`.

2 Call `os_database::save()` or `os_transaction::commit()` to automatically store all new persistent objects and all updated persistent objects in their databases.

Object retrieval is also almost completely automatic. These are the basic steps you follow to retrieve objects from a database:

1 Retrieve an initial, named entry-point object from a database using the PSE Pro API.

2 Use navigation (pointer dereferencing) to retrieve other objects from the database in just the same way that you would with regular C++ data.

When you dereference a pointer to an object that has not yet been retrieved from the database, PSE Pro retrieves the object automatically and transparently.

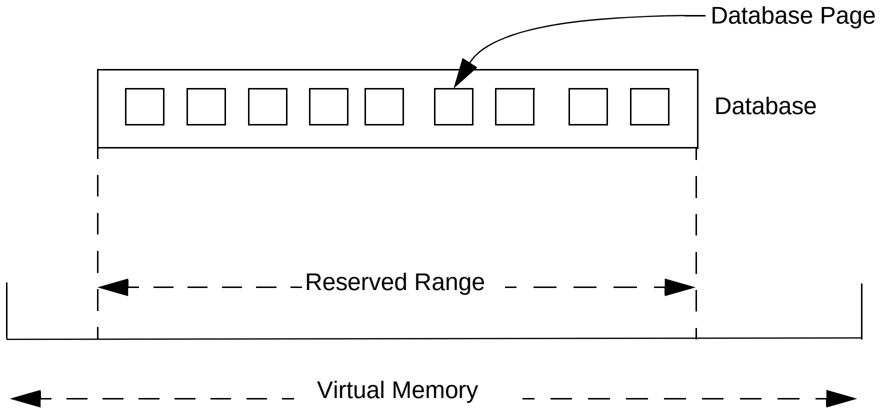
When you dereference a pointer to an object that has already been retrieved, *access to it is as fast as access to regular C++ data.* This is because objects that have already been retrieved reside in virtual memory, and pointers to them are regular pointers that can be processed at regular hardware speeds.

PSE Pro Architecture

PSE Pro uses a much simpler architecture than the ObjectStore architecture. As a result, the transfer of objects between database memory and program memory is simpler.

When PSE Pro stores a C++ object, it stores it in its native format. A database consists of pages that contain C++ objects arranged just as they would be stored in virtual memory. When an application opens a database, PSE Pro reserves a range of unmapped virtual memory addresses for all the database’s data. This means that PSE Pro ensures that the range is used for no other purpose than for the database’s pages, and that the database is mapped to a contiguous range of virtual memory.

When an object is transferred to program memory, it is not necessarily stored at the same virtual memory address it originally occupied. However, it is stored in the original location *adjusted by the offset* used to adjust the database’s pointers. That way, pointers to the object in other transferred objects are valid virtual memory pointers that refer to their original targets. Pointers that are used internally by C++, such as virtual function table pointers, might be changed.



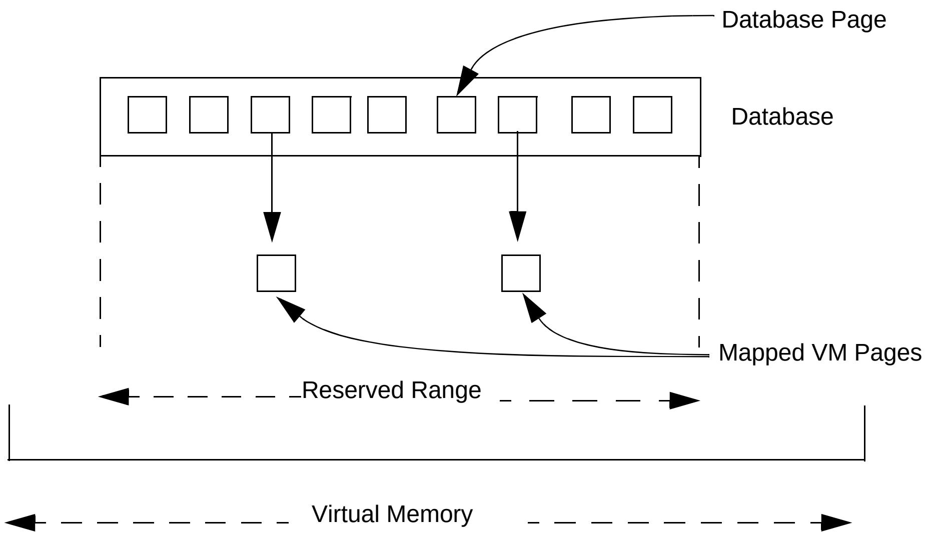
Reserving a virtual memory range effectively *assigns* each database page to a page of virtual memory, but the database pages are not yet transferred to their corresponding pages of virtual memory, and the virtual memory pages remain unmapped.

To gain initial access to a database, a program looks up an entry-point object by name, and PSE Pro transfers into virtual memory the database page that contains the object. Subsequently, the program can follow pointers contained in the transferred page.

Transferring Pages to Virtual Memory

PSE Pro detects pointers to as-yet-untransferred data by using the operating system's virtual memory mapping capability. By ensuring that any pointer to untransferred data points into an unmapped portion of virtual memory, PSE Pro ensures that the operating system signals an exception (a *read fault* or *write fault*) when a program attempts to dereference such a pointer. PSE Pro traps this signal when it occurs, and transfers into virtual memory the database page that contains the pointer's target.

PSE Pro transfers a database page by mapping a copy of it into virtual memory with read-only protection (for a read fault) or read/write protection (for a write fault). As part of the copy process, PSE Pro uses information derived from the schema object file (a file you generate to describe the classes in your database) to set pointers used internally by C++. All other pointers are copied as seen below.



Writing Pages to the Database

The first attempt by a program to update a page results in an operating system exception (a write fault). PSE Pro traps the signal, transfers the page if necessary, and changes the page's protection to read/write. Then the program proceeds with the update.

When a PSE Pro program performs a save operation on a database, PSE Pro copies to the database all pages that are mapped for read/write, and then changes their protection back to read-only. That way, the pages will not be written to the database on the next save unless they are updated again.

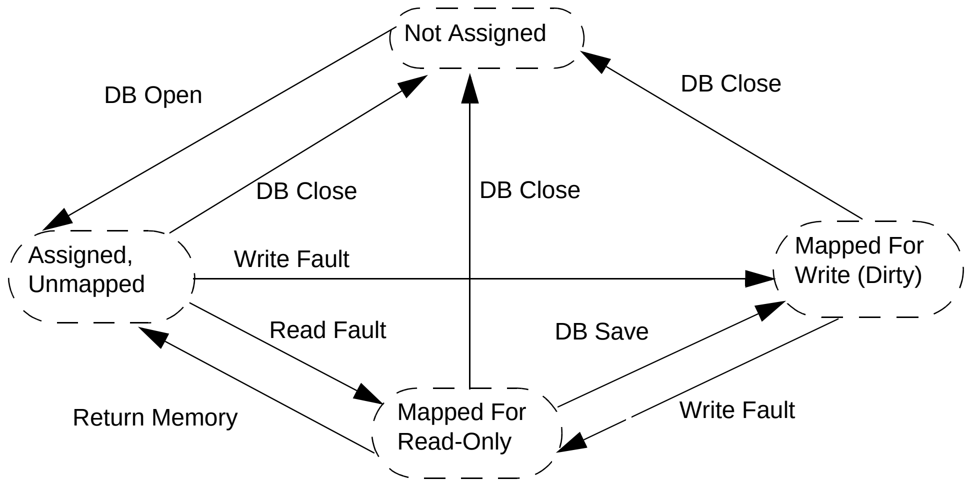
The `os_database::commit()` function in PSE Pro plays the same role as `os_database::save()`. PSE Pro failures during saves and commits are recoverable.

Unmapping and Unreserving Pages

When a database is closed, PSE Pro unmaps all its pages from virtual memory, and unreserves the database's virtual memory range.

You can also call a PSE Pro API function to explicitly unmap a page or range of pages from virtual memory, to free up swap space. In that case, the virtual memory remains reserved for the page or range of pages.

Here is a state/transition diagram that summarizes the various states of a database page:



Chapter 2 An Example Using PSE Pro

This chapter provides an example of how to use PSE Pro, along with an overview of PSE Pro features and build requirements. It discusses the following topics:

Description of the QuickStart Example

The QuickStart example is a simple example that consists of two programs. The source code for the programs is contained in `examples/quickstart` in the PSE Pro directory hierarchy. One program creates a PSE Pro database (persistent storage file) and stores two objects in it. The other program retrieves the objects and prints their data member values.

The stored objects are instances of the classes `Document` and `Person`. The `Document` class has ID and author data members, and the `Person` class has an ID data member. Although classes in a real application would have many additional members, this example illustrates all the basics of using PSE Pro.

The first program creates a new `Document` object that points to a new `Person` object (the document's author), establishes the `Document` object as a database *entry point*, and saves the database.

The second program retrieves the `Document` object via entry-point lookup, and then follows the `Document::author` pointer to the `Person` object. The `Person` object is retrieved automatically.

Also provided is a *schema source file*, which lists the classes of persistent objects used by the programs. PSE Pro uses a compiled version of this file at run time to get information about the database objects being stored and retrieved.

To build the executables, you compile the schema source file with the PSE Pro utility `pssg`, which produces the *schema object file*. You compile the application sources with your C++ compiler, and then link with the schema object file and the PSE Pro library.

Source Code for QuickStart Example

The source code for the QuickStart example, which is listed in the following sections, includes the

- QuickStart header file
- Program that creates a database and stores some objects in the database
- Program that retrieves some objects from the database
- Schema source file

QuickStart Header File

The following is the header file that both programs in the QuickStart example use:

```
/*** PDoc.h ***/

class Person;
class Document
{
public:
  int doc_id;
  Person *author;
  Document(int id, Person *a) { doc_id = id; author = a; }
};

class Person
{
public:
  int person_id;
  Person(int id) { person_id = id; }
};
```

Source Code for the Program That Stores Objects

The following is the source code for the program that creates a database and stores some objects in the database:

```
/*** create.cpp ***/

#include <os_pse/ostore.hh> // PSE Pro header file
#include <iostream>
#include "PDoc.h"

void main()
// Creates a document and a person in a new PSE Pro database.
// Sets the document to point to the person.
// Establishes the document as a database entry point.
{
  // Establish PSE Pro fault handler.
  OS_PSE_ESTABLISH_FAULT_HANDLER

  // Initialize PSE Pro.
  objectstore::initialize();

  // Create PSE Pro database (persistent storage file).
  os_database *db = os_database::create("DocDB");

  // Store a Person and a Document in the database.
  Person *a_person =
    new(db, os_ts<Person>::get()) Person(111);
  Document *a_document =
    new(db, os_ts<Document>::get()) Document(001, a_person);

  // Establish a_document as an entry point for db.
  db->create_root("doc_root")->set_value(a_document);

  // Save the database.
  db->save();

  // Close the database.
  db->close();

  cout << "Done." << endl;

  // End PSE Pro fault handler scope.
  OS_PSE_END_FAULT_HANDLER

}
```

Source Code for the Program That Retrieves Objects

The following is the source code for the program that retrieves some objects from a database:

```
/*** read.cpp ***/

#include <os_pse/ostore.hh> // PSE Pro header file
#include <iostream>
#include "PDoc.h"

void main()
// Prints Document and Author IDs.
{
  // Establish PSE  Pro fault handler.
  OS_PSE_ESTABLISH_FAULT_HANDLER

  // Initialize PSE Pro.
  objectstore::initialize();

  // Open PSE Pro database (persistent storage file).
  os_database *db = os_database::open("DocDB");

  // Look up entry-point object.
  Document *a_document = (Document*) (
    db->find_root("doc_root")->get_value()
  );

  // Navigate to author.
  Person *a_person = a_document->author;

  // Print values.
  cout << "Document ID: " << a_document->doc_id << endl ;
  cout << "Author ID: " << a_person->person_id << endl;

  // Close database.
  db->close();

  cout << "Done." << endl;

  // End PSE Pro fault handler scope.
  OS_PSE_END_FAULT_HANDLER
}
```

Schema Source File for QuickStart Example

Both programs in the QuickStart example use the following schema source file. The schema source file describes the classes that the application can store in a database.

```
/*** schema.scm ***/

// In the schema source file,
// include and mark each persistent class.

#include <os_pse/ostore.hh> // PSE Pro header file
#include "PDoc.h"

OS_MARK_SCHEMA_TYPE(Document);
OS_MARK_SCHEMA_TYPE(Person);
```

PSE Pro APIs in the Example

This section describes the functions and operators for

- Creating and Manipulating Databases on page 18
- Allocating Objects in a Database on page 19

It also provides information on

- Restrictions on Persistent Data Access on page 19
- Creating Roots and Designating Entry-Point Objects20

Creating and Manipulating Databases

A PSE Pro database is an operating system file or a stream within an OLE compound file. It contains C++ objects stored in their native C++ format. The PSE Pro API allows you to perform the following database-related tasks:

- Create a database
- Open a database
- Close a database
- Destroy a database
- Save a database
- Copy a database

In the QuickStart example, the program creates a database with this call:

```
os_database *db = os_database::create("DocDB");
```

PSE Pro uses the `os_database` class to represent a database. The `create()` function takes an argument that specifies the name of the database.

For more complete information, see Chapter 3, Managing Databases, on page 25.

Allocating Objects in a Database

You allocate and initialize memory in a database by using an overloaded C++ `new` operator, supplied by the PSE Pro API. PSE Pro also provides an implementation of the `delete` operator that you use to delete persistent objects and free persistent memory.

After you allocate persistent memory, you can use pointers to this memory in the same way you use any pointers to virtual memory. Pointers to persistent memory always take the form of regular C++ virtual memory pointers.

In the QuickStart example, the program allocates objects in the database with these calls:

```
Person *a_person =
  new(db, os_ts<Person>::get()) Person(111);
Document *a_document =
  new(db, os_ts<Document>::get()) Document(001, a_person);
```

For more complete information about using persistent `new` and `delete`, see Chapter 4, Storing and Updating Objects in a Database, on page 45.

Restrictions on Persistent Data Access

PSE Pro imposes the following restrictions on persistent data access:

- If you store a pointer to transient memory in a database object, the pointer is valid only until the database is closed. You can use fetch hooks to help you handle pointers to transient memory.
- Cross-database pointers are not allowed. A pointer in one database must not point to data in another database. ObjectStore does not have this restriction.

Creating Roots and Designating Entry-Point Objects

A database root provides a way to give an object a persistent name. This allows the object to serve as an initial *entry point* into the database. When an object has a persistent name, any process can look it up by that name to retrieve it. After you retrieve one object, you can retrieve any object related to it by using navigation, that is, by following data member pointers.

Here is the code in the QuickStart example that creates a database root:

```
db->create_root("doc_root")->set_value(a_document);
```

Each database needs only one or a small number of roots, because you usually retrieve objects through navigation. For example, suppose that you need to store an assembly in persistent memory. It is typically sufficient to name just the topmost object in the assembly. You retrieve the topmost object using name lookup, and then you navigate to the other objects in the assembly. Of course, this assumes that each component has a data member that points to a collection of its children.

Since most persistent objects are part of a network of related objects, like an assembly, you usually do not have to name or explicitly retrieve them. Use only a small number of roots. Navigating to an object is significantly faster than looking up an entry point.

For more information, see Establishing Roots as Database Entry Points on page 54 and Retrieving Roots for Database Entry on page 56 in Chapter 4.

Additional PSE Pro Features

This section provides an introduction to the following PSE Pro features:

- Transactions on page 20
- Undo/Redo Facility on page 21
- Object Iterators on page 21
- Schema Evolution on page 21
- Threads on page 22
- Fetch Hooks on page 22
- Environment Variables on page 22
- Directories Added to Search Paths on page 22

Transactions

PSE Pro for C++ supports transactions. A transaction is a logical unit of work. It groups together updates to a database in such a way that either all of those updates are performed or none of them are — even if a failure occurs in the middle of the transaction. With transactions, you are guaranteed that the database will not be left in an inconsistent, intermediate state, with some but not all of the updates stored in the database.

If a failure occurs in the middle of a transaction (for example, because of network or system failure), you can always recover your data as of the beginning of the transaction. In addition, you can explicitly abort a transaction at any time, rolling back persistent data to the beginning of the transaction.

Using transactions with PSE Pro is optional. If you want to use transactions, you first initialize the transaction facility. Once the facility is initialized, all access to databases must take place within a transaction.

For more information, see Chapter 6, Transactions, on page 79.

Undo/Redo Facility

The PSE Pro undo/redo facility allows you to undo any number of previous operations, and lets you redo any number of undone operations. PSE Pro provides an API to

- Set undo points. These define the boundaries of operations.
- Undo the database changes that were made during the previous n operations (that is, roll the database back to the nth-to-the-last undo point).
- Redo the database changes that were made during the previous n undone operations.

You can also have PSE Pro enforce a limit on the number of operations that can be undone.

You cannot undo operations performed prior to the last database save or transaction commit operation.

For more information, see Chapter 7, The Undo/Redo Facility, on page 85.

Object Iterators

Using an object cursor, an application can iterate through all objects within a database. This allows you, for example, to copy the contents of a PSE Pro database into an ObjectStore database. See Chapter 8, Object Iteration, on page 89.

Schema Evolution

The layout of objects in a PSE Pro database is identical to their layout in transient memory (heap, stack). If a class definition changes in the application, for example a data member is added to a class, the database must be evolved to be compatible with the new class definition. This is called schema evolution. See Chapter 9, Evolving Database Schemas, on page 93.

Threads

PSE Pro supports multithreaded applications. The PSE Pro thread-locking facilities ensure that PSE Pro does all necessary interlocking between threads to prevent threads from interfering with each other when within the PSE Pro run time. Note that you are responsible for coding any thread synchronization required by your application while threads are not executing within the PSE Pro library.

You do not have to do anything to activate the thread-locking facility. Thread locking is performed automatically for multithreaded applications.

For static builds, all threads that access persistent data have to be attached to and detached from PSE Pro. See *Building Applications with PSE Pro for C++* .

Fetch Hooks

With PSE Pro, you can register a function that is called whenever instances of a given class are retrieved. Fetch hooks allow you to, for example, manage pointers to transient objects.

Environment Variables

PSE Pro provides the following environment variables:

- `OS_PSE_AS_SIZE` on page 75
- `OS_PSE_AS_START` on page 75
- `OS_PSE_DEF_BREAK_ACTION` on page 75
- `OS_PSE_PAGE_SIZE` on page 76
- `OS_PSE_TRACE_ALLOC` on page 78
- `OS_PSE_VERIFY_ALLOC` on page 78

Directories Added to Search Paths

On Windows platforms the PSE Pro installation adds the following directories to the Microsoft IDE's directory search paths and to your system environment:

- Include files: `c:\odi\PSEPROC63\include`
- Executable files: `c:\odi\PSEPROC63\bin`
- Library files: `c:\odi\PSEPROC63\lib`

On UNIX systems, set the following environement variables:

- Set `OS_PSE_ROOTDIR` to `/opt/ODI/PSEPROC63`
- Set PATH to include `$OS_PSE_ROOTDIR/bin`.
- Set SHLIB_PATH to include `$OS_PSE_ROOTDIR/lib`

PSE Pro Application Requirements

The following sections describe the required components of a PSE Pro application, and how to build an application or library:

- Including the PSE Pro Header File on page 23
- Creating Schema Source Files on page 23
- Establishing Fault Handlers on page 24
- Initializing PSE Pro on page 24
- Building PSE Pro Applications and Libraries on page 24

Including the PSE Pro Header File

For an application to use any PSE Pro features, it must include the file `os_pse/ostore.hh`.

Creating Schema Source Files

Every PSE Pro application or library must be linked with an object file that contains information about the classes of persistent objects used by the application. You generate this object file from the *schema source file*. An application's *schema* consists of the classes of objects in the databases it uses. You create a schema source file as follows:

- Include the PSE Pro header file `<os_pse/ostore.hh>`.
- Include the definition of each class of object the application might store in a database, as well as the definition of each class of object in each database the application might open.
- Call the `OS_MARK_SCHEMA_TYPE()`macro for each included class, supplying the class name as argument. Do *not* put quotation marks around the class name. Enter the argument *without white space*. The macro cannot span more than *one* row in the `.scm` file.
- Give the file a name with the extension `.scm`.

Here is an example of a schema source file:

```
/*** schema.scm ***/
#include <os_pse/ostore.hh>
#include "image_map.h"
#include "image.h"
#include "document.h"
OS_MARK_SCHEMA_TYPE(image_map);
OS_MARK_SCHEMA_TYPE(image);
OS_MARK_SCHEMA_TYPE(document);
```

For classes whose definition is embedded within a `private` or `protected` section of another class's definition, use `OS_MARK_SCHEMA_NESTED_TYPE()` instead of `OS_MARK_SCHEMA_TYPE()`.

For complete information about creating schema source files, see "Chapter 2, Generating the Application Schema" in *Building Applications with PSE Pro for C++* .

Establishing Fault Handlers

PSE Pro must handle all memory access violations, because some are actually references to persistent memory in a database. On Windows, there is no function for registering a signal handler for such access violations that will also work when you are debugging a PSE Pro application (the `SetUnhandledExceptionFilter()` function does not work when you use the Visual C++ debugger).

Every PSE Pro application must put a handler for access violation at the top of every stack in the program. This normally means putting a handler in a program's `main()` or `WinMain()` function and, if the program uses multiple threads, putting a handler in the first function of each new thread.

PSE Pro provides two macros to use in establishing a fault handler:

- `OS_PSE_ESTABLISH_FAULT_HANDLER`
- `OS_PSE_END_FAULT_HANDLER`

On UNIX platforms PSE Pro can register a signal handler for such access violations and the handler needs to be registered only once using the macros `OS_PSE_ESTABLISH_FAULT_HANDLER` and `OS_PSE_END_FAULT_HANDLER`. You can also invoke the macro for every thread, as on Windows, with no ill effects.

Initializing PSE Pro

To use the PSE Pro API, you must first call the static member function `objectstore::initialize()`. The function signature is

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

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

Building PSE Pro Applications and Libraries

Follow these steps to build an application or library that uses PSE Pro:

1. Use the PSE Pro schema generator (`pssg`) to generate a schema object file from the schema source file.
2. Use your C++ compiler to compile your application or library source files.
3. Link your application or library with the schema object file and the PSE Pro library.

For more information, see *Building Applications with PSE Pro for C++* .

Chapter 3 Managing Databases

This chapter provides information about how to manage the databases you create to store your objects. It discusses the following topics:

Creating Databases

You can create databases in which to store your objects. PSE Pro uses the `os_database` class to represent your database. You create databases with `os_database::create()` or `os_database::open()`.

Creating Databases with os_database::create()

The following function creates and opens a new database as an operating system file with the specified pathname:

```
static os_database* create(
  const char* pathname,
  os_int32 prot_mode = 0664,
  os_boolean if_exists_overwrite = 0,
  os_unsigned_int32 reserve_mb_addr_space = 8
) ;
```

The following overloadings create and open a new database as a stream within the given `Istorage`:

```
static os_database* create(
  IStorage* istorage,
  const char* stream_name,
  os_unsigned_int32 istorage_mode,
  os_unsigned_int32 reserve_mb_adr_space= 8
);
static os_database* create(
  IStorage* istorage,
  const wchar_t* stream_name,
  os_unsigned_int32 istorage_mode,
  os_unsigned_int32 reserve_mb_adr_space= 8
);
```

Since `os_database::create()` is static, it has no `this` argument.

PSE Pro allows you to open multiple databases at the same time. You cannot create a database inside a transaction. If you try to, PSE Pro throws `os_err_trans`.

The following sections describe the function arguments for `os_database::create.`

pathname

Specify the name you want the new database to have. It is the only required argument. The `pathname` is an operating system pathname. It can be a relative or a rooted `pathname`. Here is an example:

```
os_database *db1 = os_database::create( "c:kendb1" ) ;
```

If you supply the `pathname` of a database that already exists, PSE Pro throws `os_err_database_exists`, unless `if_exists_overwrite` is nonzero.

If you supply the `pathname` of a file that is not a PSE Pro database, PSE Pro throws `os_err_not_a_database`.

If `pathname` is not a valid operating system name, PSE Pro throws `os_err_invalid_pathname`.

The function takes into account local network mount points when interpreting the pathname, so the pathname can refer to a database on a remote host.

PSE Pro cannot recognize when two different pathnames refer to the same database. Different pathnames are always assumed to refer to different databases.

prot_modes

Specify the *protection mode* of the new database. This controls access permissions for the new database.

A protection mode is an octal number constructed from the OR of the following modes (note that *execute* is meaningful only for directories):

Mode	*Description*
`400`	Read by user
`200`	Write by user
`100`	Execute (search in directory) by user
`040`	Read by group
`020`	Write by group
`010`	Execute (search) by group
`004`	Read by others
`002`	Write by others
`001`	Execute (search) by others

This argument must begin with a zero (0) to indicate that it is an octal number. The mode defaults to `0664.` This specifies read and write permission for user and group, and read-only by others. For example:

```
os_database *db1 = os_database::create("\kendb1", 0666) ;
```

specifies read and write permission for user, group, and other.

if_exists_overwrite

Specify whether or not to throw an exception when overwriting an existing database. Specify a nonzero value for the argument to overwrite any existing database with the specified name to avoid throwing an exception. For example:

```
os_database *db1 = os_database::create("\kendb1", 0666, 1);
```

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

reserve_mb_add_space

Specify the number of megabytes of address space to reserve for the database's data for the current session (the period from database open to database close). The default is 8 MB.

Each time you initialize PSE Pro, it reserves a range of virtual addresses for the current application. This range is called the *persistent storage region* (PSR).

Each time you open a database, PSE Pro reserves a subrange of the PSR for that database. This range accommodates all the database's data, or the amount specified by *`reserve_mb_addr_space`*, whichever is larger.

If you try to allocate space for new data in a database, and the data does not fit in the database's reserved range, the range must be extended. The added address space must be contiguous with the already reserved range. If possible, the address range is extended automatically, even if this requires extending the PSR as well. But this might not be possible for two reasons:

- The database's range already approaches the end of the PSR, and the PSR cannot be extended.
- You are using PSE Pro with multiple databases open at once, and other databases have already reserved some of the required addresses.

If it is not possible to extend the database's reserved range, PSE Pro throws `os_err_address_space_exhausted`.

To help you avoid getting this exception, `database::create()` and `database::open()` allow you to specify the amount of address space to reserve at open time. If you reserve enough extra address space to avoid the need to extend the range, you can avoid getting `os_err_address_space_exhausted`.

istorage

Specify the name of the `IStorage` object.

stream_name

Specify the name of the stream to create. The `char*` ***`stream_name`*** in the first overloading must be a normal multibyte string. The `wchar_t*` ***`stream_name`*** argument in the second overloading, is Unicode compatible.

istorage_mode

Specify a storage mode using a bit field taken from the STGM enumeration:

```
STGM_DIRECT  0x00000000L
STGM_TRANSACTED  0x00010000L
STGM_SIMPLE  0x08000000L
STGM_READ  0x00000000L
STGM_WRITE  0x00000001L
STGM_READWRITE  0x00000002L
STGM_SHARE_DENY_NONE  0x00000040L
STGM_SHARE_DENY_READ  0x00000030L
STGM_SHARE_DENY_WRITE  0x00000020L
STGM_SHARE_EXCLUSIVE  0x00000010L
STGM_PRIORITY  0x00040000L
STGM_DELETEONRELEASE  0x04000000L
STGM_CREATE  0x00001000L
STGM_CONVERT  0x00020000L
STGM_FAILIFTHERE  0x00000000L
STGM_NOSCRATCH  0x00100000L
```

The mode bits are passed as is to `IStorage::CreateStream()` and `IStorage::OpenStream()`.

Controlling Database Size

Database files cannot shrink. When you delete an object in a database, the space goes to free space. PSE Pro reuses it for a subsequent allocation. If you copy a database, the target database has the same size as the source database. However, PSE Pro does work well with compressed disks, so that free space does not use any storage bytes.

When you open a database, reserve at least enough to accommodate the maximum amount of data the database will have between the time of the open and the time at which it is closed. You might have to reserve more address space if data is sometimes deleted from the database.

Although it is rarely necessary, you can also adjust the size and location of the application's PSR.

The Return Value

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

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

Creating Databases with os_database::open()

Typically, you call `os_database::create()` to create a database, but you can also use the function `os_database::open()` to create a database as an operating system file. This function requires you to specify the pathname of the database you want to open. You can specify an optional argument to direct the function to create a database if a database with the specified pathname does not exist.

Opening Databases

Before you can read or write data, you must open the database in which it resides. When you create a database with `os_database::create()`, PSE Pro opens it. However, to open an existing database, you use `os_database::open()` described in this section:

os_database::open()

If you want to open a database previously created as an operating system file, use the following function:

```
static os_database *open(
  const char* pathname,
  os_boolean read_only = 0,
  os_int32 create_mode = 0
  os_unsigned_int32 reserve_n_times_db_size = 0
);
```

If you want to open a database previously created as a stream within a specified `Istorage`, use one of the following functions:

```
static os_database* open(
  IStorage* istorage,
  const char* stream_name,
  os_unsigned_int32 istorage_mode,
  os_unsigned_int32 reserve_n_times_db_size = 0
);
```

```
static os_database* open(
  IStorage* istorage,
  const wchar_t* stream_name,
  os_unsigned_int32 istorage_mode,
  os_unsigned_int32 reserve_n_times_db_size = 0
);
```

PSE Pro allows you to open multiple databases at the same time.

The following sections describe the arguments for `os_database::open()`.

pathname

Specify the name of the database you want to open. It is the only required argument. The pathname is an operating system pathname. It can be a relative or rooted pathname. Variants of a pathname are not interpreted as referring to the same file. The following example opens the database `db1`:

```
os_database *db1 = os_database::open( "db1" ) ;
```

If there is no file with the specified pathname, PSE Pro throws `os_err_database_not_found`, unless *`create_mode`* is nonzero.

If you supply the pathname of a file that is not a PSE Pro database, PSE Pro throws `os_err_not_a_database`. PSE Pro throws the same exception if you attempt to open a file that resulted from an aborted database creation.

If the pathname is not a valid operating system pathname, PSE Pro throws `os_err_invalid_pathname`.

The function takes into account local network mount points when interpreting the pathname, so the pathname can refer to a database on a foreign host.

read_only

Specify whether the database is to be opened for read-only or update. A nonzero integer (true) indicates read-only, and `0` (false) indicates that write access is allowed. The default is `0`. The following example opens the database `kendb1` with read-only access.

```
os_database *db1 = os_database::open("kendb1", 1) ;
```

Multiple processes can have a given database open at the same time, if they all open the database for read-only. While a process has a database opened for write, no other process can open the database.

If you attempt to write to a database that has been opened for read-only, PSE Pro throws `os_err_write_permission_denied`.

The class `os_database` also provides functions for determining a database's open status. See Obtaining Information About a Database on page 38.

create_mode

Specify the mode of a new database to be created if a file with the specified pathname does not exist.

If *`create_mode`* is `0`, and there is no file with the specified name, PSE Pro throws `os_err_database_not_found`.

If *`create_mode`* and *`read_only`* are both nonzero, and there is no database with the specified name, PSE Pro throws `os_err_write_permission_denied`. You cannot open a new database for read-only.

Do not create databases inside a transaction. If you try to, PSE Pro throws `os_err_trans`.

reserve_n_times_db_size

Specify the amount of address space to reserve for the database's data for the current session (the period from database open to database close). If the value of the argument is nonzero, it specifies a multiple of the database size. If this argument is `0` (the default), the following occurs:

- If the database size is less than 8 MB, PSE Pro reserves 8 MB.
- If the database size is more than 8 MB but less than 64 MB, PSE Pro reserves twice the database size.
- If the database size is greater than 64 MB, PSE Pro reserves address space that is 8 MB larger than the database.

See `reserve_mb_add_space` on page 27 for information on deciding how much address space to reserve.

istorage

Specify the name for an `IStorage` object.

stream_name

Specify the name of the stream to open. The `char*` *`stream_name`* in the first overloading must be a normal multibyte string. The `wchar_t*` *`stream_name`* argument in the second overloading is Unicode compatible.

istorage_mode

Specify a storage mode using a bit field taken from the STGM enumeration:

```
STGM_DIRECT  0x00000000L
STGM_TRANSACTED  0x00010000L
STGM_SIMPLE  0x08000000L
STGM_READ  0x00000000L
STGM_WRITE  0x00000001L
STGM_READWRITE  0x00000002L
STGM_SHARE_DENY_NONE  0x00000040L
STGM_SHARE_DENY_READ  0x00000030L
STGM_SHARE_DENY_WRITE  0x00000020L
STGM_SHARE_EXCLUSIVE  0x00000010L
STGM_PRIORITY  0x00040000L
STGM_DELETEONRELEASE  0x04000000L
STGM_CREATE  0x00001000L
STGM_CONVERT  0x00020000L
STGM_FAILIFTHERE  0x00000000L
STGM_NOSCRATCH  0x00100000L
```

The mode bits are passed as is to `IStorage::CreateStream()` and `IStorage::OpenStream()`.

Exceptions

If the specified database is opened for write by another process, PSE Pro throws `os_err_database_is_locked`.

If the database is already opened by the current process, PSE Pro throws `os_err_database_in_use`.

If you try to open a file database that requires recovery, PSE Pro throws `os_err_database_needs_recovery`.

Return Value

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

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

Closing Databases

You close a database by calling `os_database::close()`. The function signature is

```
void close() ;
```

This makes the database's data inaccessible, and deletes the instance of `os_database` pointed to by `this`. PSE Pro unmaps all the database's data from virtual memory. Of course, this does not delete the database itself.

When you close a database, PSE Pro does not automatically save the state of the database.

Do not call this function inside a transaction. If you try to, PSE Pro throws `os_err_trans`.

Destroying Databases

You can delete a database with `os_database::destroy()`. The function signature is

```
void destroy() ;
```

The `destroy()` function closes and deletes the database represented by this `os_database` object. PSE Pro deletes the `os_database` object as well.

Example

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

If the database is not opened for write, PSE Pro throws `os_err_write_permission_denied`.

Do not call this function inside a transaction. If you try to, PSE Pro throws `os_err_trans`.

Saving Databases

To make your changes to a database permanent, call `os_database::save()`. To save a database under a different location, call `os_database::save_as()`. You can also call `os_transaction::commit()` to make database changes permanent.

os_database::save()

The function signature for `os_database::save()` is

```
void save() ;
```

After the save operation, the database's data, including your latest changes, resides safely on disk. Once saved, your data is recoverable as of the time of the last save operation.

If a failure occurs during a save, your data, as of the previous save, is automatically recoverable. The recovery facility allows you to determine if a failure occurred during a save operation, and it allows you to restore a file database, if necessary, to its state as of the last save.

Exceptions

PSE Pro throws `os_err_write_permission_denied` when `save()` is called if the database is opened for read-only.

PSE Pro throws `os_err_database_transient` if the database is a transient database. PSE Pro throws `os_err_trans` if you call this function inside a transaction.

os_database::save_as()

When you want to save a database as an operating system file, the function signature for `os_database::save_as()` is

```
void save_as(
  const char* dest_pathname,
  os_int32 mode = 0664,
  os_boolean if_exists_overwrite = 0
) ;
```

When you want to save the database as a stream within a specified `Istorage`, the function signature is

```
void save_as(
  IStorage* dest_istorage,
  const char* dest_istream_name,
  os_unsigned_int32 istorage_mode
);
```

```
void save_as(
  IStorage* dest_istorage,
  const wchar_t* dest_istream_name,
  os_unsigned_int32 istorage_mode
);
```

The following sections describe the arguments passed to the various overloadings of the `os_database::save_as()` function.

dest_pathname

Specify the name of the new database. The `save_as()` function copies the database referred to by the implied `this` to `dest_pathname`, and saves the latest changes to the new database. After `save_as()` completes, `this` represents the destination database, and the destination database is open. The source database is closed, and no outstanding changes are saved to it.

if_exists_overwrite

Specify a nonzero value for the argument *`if_exists_overwrite`* to overwrite any existing database with the specified name to avoid throwing an exception. For example:

```
os_database *db1 = os_database::create("\kendb1", 0666, 1);
```

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

dest_istorage

Specify the name of an `IStorage` object.

dest_istream_name

Specify the stream within the specified `IStorage`. The `save_as()` function copies the database associated with `this` to *`dest_istream_name`* within *`dest_istorage`* and saves the latest changes to the new database. After `save_as()` completes, `this` represents the destination database, and the destination database is open. The source database is closed, and no outstanding changes are saved to it.

The `char* stream_name` in the first overloading must be a normal multibyte string. The `wchar_t*` *`dest_istream_name`* argument in the second overloading is Unicode compatible.

istorage_mode

Specify a storage mode using a bit field taken from the STGM enumeration:

```
STGM_DIRECT  0x00000000L
STGM_TRANSACTED  0x00010000L
STGM_SIMPLE  0x08000000L
STGM_READ  0x00000000L
STGM_WRITE  0x00000001L
STGM_READWRITE  0x00000002L
STGM_SHARE_DENY_NONE  0x00000040L
STGM_SHARE_DENY_READ  0x00000030L
STGM_SHARE_DENY_WRITE  0x00000020L
STGM_SHARE_EXCLUSIVE  0x00000010L
STGM_PRIORITY  0x00040000L
STGM_DELETEONRELEASE  0x04000000L
STGM_CREATE  0x00001000L
STGM_CONVERT  0x00020000L
STGM_FAILIFTHERE  0x00000000L
STGM_NOSCRATCH  0x00100000L
```

The mode bits are passed as is to `IStorage::CreateStream()` and `IStorage::OpenStream()`

Exceptions

If an exception is thrown during a `save_as()` operation, you are responsible for catching the exception and retrying the operation.

If the source database is opened for read-only or has read-only file system protection, PSE Pro throws `os_err_write_permission_denied`.

If the source database is the transient database, PSE Pro throws `os_err_database_ transient`.

Do not call `os_database::save_as()` inside a transaction. If you try to, PSE Pro throws `os_err_trans`.

Copying Databases

You can copy a database in two ways:

- `os_database::copy()` on page 36
- `ospsecp.exe` on page 37

Additionally, you can duplicate a database under another name using `os_ database::save_as()`. See `os_database::save_as() on page 34`.

os_database::copy()

The function signature for copying a database to a pathname is

```
void copy(
  const char* dest_pathname,
  os_int32 prot_mode = 0664,
  os_boolean if_exists_overwrite = 0
) const;
```

The following are the function signatures for copying a database to a stream within a specified `Istorage`:

```
void copy(
  IStorage* dest_istorage,
  const char* dest_istream_name,
  os_unsigned_int32 istorage_mode
) const;
```

```
void copy(
  IStorage* dest_istorage,
  const wchar_t* dest_istream_name,
  os_unsigned_int32 istorage_mode
) const;
```

For all overloadings, the implicit `this` specifies the source of the copy. The following sections describe the arguments passed to the `copy()` function.

dest_pathname

Specifies the target.

prot_mode

Specifies the protection mode of the new database.

if_exists_overwrite

Specifies whether the copy can overwrite an existing file. If a file named `dest_pathname` already exists and `if_exists_overwrite` is nonzero, the copy overwrites the file, provided the file is not a PSE Pro database that is currently in use.

dest_istorage

Specify the name of the `IStorage` object.

dest_istream_name

Specify the stream within the specified `IStorage`. The `copy()` function copies the database associated with `this` to *`dest_istream_name`* within *`dest_istorage`*.

The `char* stream_name` in the first overloading must be a normal multibyte string. The `wchar_t*` *`dest_istream_name`* argument in the second overloading is Unicode compatible.

istorage_mode

The *`storage_mode`* argument specifies the Istorage mode of the new database.

Exceptions

If a file named `dest_pathname` already exists and `if_exists_overwrite` is 0, PSE Pro throws `os_err_database_exists`.

If the source database is opened for write by another process, PSE Pro throws `os_err_database_is_locked`.

If a database named *`dest_pathname`* already exists and is opened by the current process, PSE Pro throws `os_err_database_in_use`.

If the database is a transient database, PSE Pro throws `os_err_database_transient`.

During the copy, unmapped pages of the source database are not mapped into memory. The source database is not altered in any way. Do not call `copy()` inside a transaction. If you try to, PSE Pro throws `os_err_trans`.

ospsecp.exe

You can use the `ospsecp` utility to copy a database. The format for the command is

```
ospsecp source-pathname target-pathname
```

The command copies the source database to the target using the following call to `os_database::copy()`:

```
source-db-ptr.copy(target-db-ptr, 0644, 1)
```

Obtaining Information About a Database

This section describes functions of the `os_database` class you use to obtain the following information about a database:

- Is the Database Open? on page 38
- Is the Database Open for Read-Only? on page 38
- Is the Database Open for Write Access? on page 38
- What Is the Size of the Database? on page 38
- What Is the Pathname of an Open Database? on page 39
- Has the Database Been Modified? on page 39

Is the Database Open?

```
static os_boolean is_open(os_database* db) ;
```

This function returns nonzero (true) if the specified database is open, and `0` (false) otherwise. If there is no file with the specified pathname, PSE Pro throws `os_err_database_not_found`. If you supply the pathname of a file that is not a valid PSE Pro database, PSE Pro throws `os_err_not_a_database`. If the pathname is not a valid operating system pathname, PSE Pro throws `os_err_invalid_pathname`.

The function takes into account local network mount points when interpreting the pathname, so the pathname can refer to a database on a remote host.

Is the Database Open for Read-Only?

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

This function returns nonzero (true) if this database is opened for read-only by the current process; returns `0` (false) otherwise.

Is the Database Open for Write Access?

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

This function returns nonzero (true) if this database is opened for write access by the current process; it returns `0` (false) otherwise.

Example

```
os_database* db1;
...
if (db1->is_writable())
  db1->close();
```

What Is the Size of the Database?

```
os_unsigned_int32 size() const;
```

This function returns the size of `this` database in bytes.

What Is the Pathname of an Open Database?

```
char* get_pathname() const;
```

This function returns the pathname of `this` database. The returned `char*` points to an array allocated on the heap by this function. It is the caller's responsibility to deallocate the array when it is no longer needed.

Has the Database Been Modified?

```
os_boolean os_database::is_modified() const;
```

This function returns nonzero (true) if `this` database has been modified since the last `os_database::save()` operation. Otherwise, this function returns `0` (false).

Recovering Databases

For databases that are operating system files, you can use `os_database::needs_recovery()` or the `ospserecov` utility to determine if a database needs recovery.

Call the `os_database::recover()` function to restore a database to a consistent state. Logging is the mechanism that enables recovery during save operations. If you want to improve performance, you can disable logging, but it will prevent recovery from failures during save operations.

For databases that are streams within OLE compound files, use `TRANSACTED` mode.

Function for Determining If a Database Needs Recovery

Call the `os_database::needs_recovery()` function to determine if a database needs to be recovered. The function signature is

```
static os_boolean needs_recovery(const char* pathname);
```

This function returns nonzero (true) if a failure occurred during a transaction commit, or, if transactions were not initialized, during save.

PSE Pro throws `os_err_database_not_found` if there is no file with the specified pathname.

PSE Pro throws `os_err_not_a_database` if you supply the pathname of a file that is not a PSE Pro database.

PSE Pro throws `os_err_invalid_pathname` if the pathname is not a valid operating system pathname.

Utility for Determining If a Database Needs Recovery

Another way to determine if a database needs recovery is to invoke the `ospserecov` utility. The format is

```
ospserecov -c database_pathname
```

The `-c` option instructs the utility to check whether recovery is required. The value you specify for *`database_pathname`* can be absolute or relative.

The utility calls `os_database::needs_recovery()` to determine if recovery is required. The utility sends a message to standard output that indicates whether the specified database requires recovery.

Example

```
ospserecov -c foo2.db
```

If you omit the `-c` option, the utility restores the database to its state as of the last successful commit or (if transactions were not initialized at the time of the failure) as of the last successful save operation. If the database does not need recovery, a message is sent to standard error. The utility calls `os_database::recover()` to perform the recovery. For example,

```
ospserecov foo2.db
```

Recovering a Database

You can recover a database with the `os_database::recover()` function. The function signature is

```
static void recover(const char* pathname);
```

If `os_database::needs_recovery()` returns `1` for the given database, this function restores the database to its state as of the last successful commit or, if transactions were not initialized at the time of the failure, as of the last successful save operation.

If you call this function inside a transaction, PSE Pro throws `os_err_trans`.

If there is no file with the specified pathname, PSE Pro throws `os_err_database_not_found`.

If you supply the pathname of a file that is not a PSE Pro database, PSE Pro throws `os_err_not_a_database`.

If the pathname is not a valid operating system pathname, PSE Pro throws `os_err_invalid_pathname`.

If the log file is not found or is not compatible with the database being recovered, PSE Pro throws `os_err_database_recovery`. This can happen if logging was disabled at the time of the failure or if the database or the log file has been moved since the failure. If you move them, the database and log file must remain in the same directory as one another, and the name of the log file must be the database name plus the suffix `.log` (or, on FAT, it must be the database name with `.log` replacing the database file name extension). Otherwise, this exception is thrown. This exception is also thrown if the file so named was not the log file for the specified database at the time of the failure.

Disabling Logging

```
objectstore::disable_logging()
```

This function disables logging, which is enabled by default. If logging is disabled, failures during saves are not recoverable. However, disabling logging can improve performance and reduce disk consumption. When logging is enabled, before-images of each written page are temporarily stored on disk before each save operation. Do not use transactions if you want logging disabled. If you have disabled logging, `os_transaction::initialize()` reenables it. If you call `disable_logging()` after calling `os_transaction::initialize()`, the function has no effect and logging remains enabled.

Checking for Data Corruption

You can use PSE Pro to detect some forms of data corruption in internal data structures. Your options for doing this are

- Checking Databases with the ospseverifydb Utility
- Checking Databases with the self_check() Functions

Checking Databases with the ospseverifydb Utility

You can run the `ospseverifydb` utility to check a PSE Pro database. When PSE Pro checks a database, it verifies that

- There are no transient pointers.
- Persistent pointers point to objects within the database.
- Metadata values are valid.

The utility displays a message about anything it finds that is invalid. The utility does not terminate if it finds illegal pointers. The utility does terminate if

- It finds metadata corruption.
- Typespecs are missing for classes that need to be verified.

The command line format for the `ospseverifydb` utility is as follows:

```
ospseverifydb db [-schema_lib schema_library...]
```

Replace *`db`* with the path of the PSE Pro database that you want to verify. If the database you want to verify contains only built-in types and no user-defined types, you need to specify only the database path.

If the database you want to verify contains any user-defined types, you must specify the `-schema_lib` option and replace *`schema_ library`* with the path for one or more schema libraries. Specify the schema library or libraries that contain the definitions of the user-defined types in the database you want to verify. If you do not specify at least one schema library, the utility verifies C++ built-in types until it finds a user-defined type. At which point the utility throws `os_err_umarked_type` exception and terminates.

For applications that run on Windows, a schema library is a dll. For applications that run on UNIX, a schema library is a shared library. You use the `pssg` utility to generate schema libraries. For information about generating schema libraries, see *Building PSE Pro C++ Applications*, Running the Schema Generator.

Checking Databases with the self_check() Functions

You can call `os_database::self_check()` to check a single database or `objectstore::self_check()` to check all open databases. When PSE Pro checks a database, it verifies that

- There are no transient pointers.
- Persistent pointers point to objects within the database.
- Metadata values are valid.
- Virtual table pointers are valid — However, if you set the `b_skip_vtbls` argument to true, then the function does not check whether virtual table pointers are valid.

The operation does not terminate if it finds illegal pointers. The operation does terminate if

- It finds metadata corruption.
- Typespecs are missing for classes that need to be verified.

A side effect of these functions is that all pages are mapped into memory. However, if you call `self_check()` with `b_skip_vtbls` set to true, then all pages are unmapped before `self_check()` returns.

os_database::self_check()

```
void self_check(os_boolean b_skip_vtbls = 0);
```

Checks the internal data structures of `this` database. If corruption is detected, PSE Pro throws the `os_err_heap_corruption` exception. The heap corruption can be due to a memory corruption problem in user code or a product defect.

Set the `b_skip_vtbls` argument to true when your program performs only database verification and the class definitions are not linked into the program. In such cases, the virtual table pointers are not available to the program and so their verification needs to be skipped, which is accomplished by setting `b_skip_vtbls` to true.

objectstore::self_check()

```
static void self_check(os_boolean b_skip_vtbls = 0);
```

Checks the internal data structures of all open databases, as well as internal structures not associated with a particular database. If corruption is detected, PSE Pro throws `os_err_heap_corruption`. The heap corruption can be due to a memory corruption problem in user code or a product defect.

Set the `b_skip_vtbls` argument to true when your program performs only database verification and the class definitions are not linked into the program. In such cases, the virtual table pointers are not available to the program and so their verification needs to be skipped, which is accomplished by setting `b_skip_vtbls` to true.

Finding Invalvid Pointers After Relocation

The `obectstore::set_check_illegal_pointers()` function indicates whether you want PSE Pro to find any invalid pointers in a page after relocation. The function signature is

```
void set_check_illegal_pointers(os_boolean yes_no);
```

After you call this function with a true value, then after mapping a page into memory and performing relocation, PSE Pro checks the pointers in the relocated page to determine whether they are valid. PSE Pro displays a message that indicates any invalid pointers, and then continues processing. PSE Pro does not fix any invalid pointers.

The `OS_PSE_CHK_ILLEGAL_PTR` environment variable serves the same purpose as a call to the `set_check_illegal_pointers()` function.

The default value for the function call and for the environment variable is false.

Dumping Databases

You can dump the contents of a database with `os_database::dump_contents()`. The function signature is

```
void dump_contents();
```

This function dumps the contents of `this` database to standard output. The dump includes allocated as well as free blocks. For each block, the function displays the address, size, type (if allocated), tag value, array count, and block size.

The program calling this function must mark all the classes in the database that would have to be marked by an application retrieving the classes' instances in the usual way (that is, using a root and navigation as opposed to `dump_contents()`).

Chapter 4 Storing and Updating Objects in a Database

To store objects in a database, you must create them with PSE Pro's persistent `new` function. Once created, you need to know how to manipulate persistent objects, but also when and how to use a transient database. To help you perform many operations on your database objects, this chapter discusses the following topics:

Persistent New

The heart of the PSE Pro API is a special overloading of the global function `::operator new()`, known as *persistent* `new`. Allocating data with persistent `new` is a lot like allocating data with regular `new`. The difference is that data allocated with regular `new` is *transient*. That is, it disappears after the allocating application terminates. Data allocated with persistent `new` resides in stable storage, in a database, until explicitly deleted. You can use persistent `new` to allocate persistent instances of built-in types such as `int` or `char*`, as well as classes that you define. Both nonarray and array overloadings provide you with some additional flexibility in the objects you create.

To prepare a class so that its instances can be stored in a PSE Pro database, you might have to modify some member function implementations. If a member function allocates any new objects, you must decide whether these new objects should be allocated in a database. If so, you must change the allocation to use persistent `new` as defined by the PSE Pro API.

Persistent `new` takes arguments for

- An `os_database*` that indicates the database in which to allocate the new object
- An `os_typespec*` that specifies the type of the new object

Creating Nonarrays with Persistent new

Use the following overloading of `::operator new()` to create scalar (nonarray) objects in database memory:

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

`db` specifies the database in which the new object is to be stored. If you specify a transient database, PSE Pro allocates the new object on the transient heap.

`typespec` specifies the type of the new object. If you are creating an instance of a class, *`C`*, use the static member function `os_ts<C>::get()`. If you are creating an instance of a fundamental type, use a member of os_typespec. If *`db`* specifies a transient database, you can supply `0` for *`typespec`*.

Additional overloadings of persistent `new` are provided for creating arrays and supplying a clustering hint.

As with ordinary `new`, persistent `new` returns a pointer to the created object. Before the constructor is called, PSE Pro initializes database memory allocated by `new` with zeros.

Restrictions

If you store a pointer to transient memory in a persistent object, the pointer is valid only until you close the database.

Cross-database pointers are not allowed. A pointer in one database must not point to data in another database.

Example

```
image_map *an_image_map =
  new( db1, os_ts<image_map>::get() ) image_map(111) ;
```

In this example, the new object is stored in the database pointed to by `db1`.

Exceptions

If *`db`* or *`typespec`* is `0`, or if the typespec size does not match the size of the type being allocated, PSE Pro throws `os_err_operator_new_args`.

If the program allocates a class instance and the size of the instance according to the program's class definition is not the same as the size of instances of the class stored in the database, PSE Pro throws `os_err_alloc`.

If new data does not fit into the persistent storage region, the range of virtual memory addresses reserved for persistent data, PSE Pro throws `os_err_address_ space_exhausted`.

If heap corruption is detected during allocation, PSE Pro throws `os_err_heap_ corruption`. The heap corruption can be due to a memory corruption problem in user code or a product defect.

Creating Arrays with Persistent new

Use the following overloading of `::operator new()` to create arrays of objects in database memory:

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

`db` specifies the database in which to store the new array. If you specify a transient database, PSE Pro allocates the new array on the transient heap.

`typespec` specifies the type of the elements in the new array. If the elements are instances of a class, *`C`*, use the static member function `os_ts<`*`C`*`>::get()`. If the elements are instances of a fundamental type, use a member of os_typespec. If *`db`* specifies a transient database, you can supply `0` for `typespec`.

`how_many` specifies the number of elements in the array.

As with ordinary `new`, persistent `new` returns a pointer to the new array. Before the constructor for the new array is called, PSE Pro initializes database memory allocated by `new` with zeros.

Additional overloadings of persistent `new` are provided for creating nonarrays and supplying a clustering hint.

Restrictions

If you store a pointer to transient memory in a database object, the pointer is valid only until you close the database.

Cross-database pointers are not allowed. A pointer in one database must not point to data in another database.

Example

```
image* some_images = new( db1, os_ts<image>::get(), 10 ) image[10];
```

This call creates an array of ten `image` objects.

An example of calling new from a constructor appears later in this chapter.

Exceptions

If *`db`*, *`typespec`*, or *`how_many`* is `0`, or if the typespec size does not match the size of the type being allocated, PSE Pro throws `os_err_operator_new_args`.

If *`how_many`* is not the same as the array size specified in the brackets, PSE Pro throws `os_err_alloc`.

If the program allocates a class instance and the size of the instance according to the program's class definition is not the same as the size of instances of the class stored in the database, PSE Pro throws `os_err_alloc`.

If new data does not fit into the persistent storage region, the range of virtual memory addresses reserved for persistent data, PSE Pro throws `os_err_address_space_exhausted`.

If heap corruption is detected during allocation, PSE Pro throws `os_err_heap_corruption`. The heap corruption can be due to a memory corruption problem in user code or a product defect.

Clustering Objects in a Database

Clustering refers to how persistent data is arranged on disk, particularly with regard to what data is close to or *clustered with* what other data. Because PSE Pro transfers persistent data a page at a time, controlling clustering can sometimes improve application performance. By clustering together objects that are used together, you can sometimes reduce the number of pages transferred between database and program memory.

This section discusses the following topics:

- About Clustering on page 48
- Factors to Consider When Determining Clusters on page 49
- Providing a Clustering Hint on page 49

About Clustering

You should try to cluster objects to reduce the total number of pages that contain data needed in one session (the period from database open to database close). If all the objects in a database are needed, the clustering does not matter.

For example, consider a database that contains a mail archive and an application that allows users to browse the archive's subject lines and display selected message bodies. If a typical session uses many subject lines and few message bodies, the application might benefit from a clustering that stores the subject lines together. That way the subject lines are spread out over fewer pages than if each subject line were stored with its corresponding message body.

The effect of clustering on transfers to the database is similar to its effect on transfers from the database. If updated objects are clustered together, fewer pages will be transferred upon save or transaction commit than if updated objects are spread out over a large number of pages. You can control clustering in the following ways:

- Controlling (temporal) allocation order: In the absence of deletions, PSE Pro uses high-water-mark allocation.
- Providing a clustering hint to persistent `new`: If there are deletions, PSE Pro tries to reuse freed database memory. With a special overloading of persistent `new`, you can direct PSE Pro to try to cluster a new object with a specified object.

Factors to Consider When Determining Clusters

Different operations sometimes benefit from different clusterings. Use the clustering that provides the greatest overall benefit, taking into account the following factors for each operation:

- How much does the clustering benefit or hurt the operation?
- How frequently is the operation performed?
- How time critical is the operation?

Providing a Clustering Hint

Use the following overloading of `::operator new()` to provide a clustering hint when creating scalar (nonarray) objects in database memory:

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

Use the following overloading of `::operator new()` for providing a clustering hint when creating arrays of objects in database memory:

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

`where` specifies an object with which PSE Pro tries to cluster the new object. In any event, PSE Pro stores the new object in the same database as the specified object. If you specify a transient object, PSE Pro allocates the new object on the transient heap. The rest of the arguments are the same as those for the other persistent `new` overloadings.

Example

```
delete msg2->subject_line;
char *new_subj_line = "changed subject line";
msg2.subject_line = new(
  msg1->subject_line,
  os_typespec::get_char() ,
  strlen(new_subj_line)+1
) char[strlen(new_subj_line)+1];
strcpy(msg2->subject_line, new_subj_line);
```

Deleting Objects

To delete persistent objects, use the persistent `delete` just as you would use the C++ `delete` operator to delete transient objects. Deleting transient objects requires a separate function also described here.

::operator delete()

```
void operator delete() ;
```

Deletes the specified object from storage. The argument must point to the beginning of an object's storage.

If heap corruption is detected during deallocation, PSE Pro throws `os_err_heap_corruption`. The heap corruption can be due to a memory corruption problem in user code or a product defect.

If you define `::operator delete()`, overriding the PSE Pro definition, use `objectstore::free_memory()` on page 50 instead.

If you do not define `::operator delete()`, but want application-specific delete processing for *transient* objects only, register a function to do the special processing with `objectstore::set_transient_delete_function()` on page 50.

objectstore::free_memory()

```
static void free_memory(void*);
```

Deletes the specified object from storage. This function calls the PSE Pro delete operator.

objectstore::set_transient_delete_function()

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

Registers a function that is called by the PSE Pro overloading of `::operator delete()`, when performed on transient objects. You can unregister a function by passing `0` to `set_transient_delete_function()`.

Using Typespecs

You specify a typespec as an argument to persistent `new`. A typespec represents a particular type, such as `char`, `image`, or `image*`. Typespecs for fundamental and pointer types are instances of the class `os_typespec`. Typespecs for classes are instances of instantiations of the class template `os_ts<T>`, which is derived from `os_typespec`. The typespec for the class `image`, for example, is an instance of the class `os_ts<image>`.

This section discusses the following topics:

- Typespecs for Classes on page 51
- Typespecs for Fundamental and Pointer Types on page 51
- `os_typespec::get_name()` on page 52
- `os_typespec::get_typespec()` on page 52
- `os_typespec::get_size()` on page 52
- `os_typespec::needs_relocation()` on page 52

Typespecs for Classes

For every class, *C*, with instances stored in a database, you can use the static function `os_ts<C>::get()` to retrieve a typespec for the class. The signature is:

```
static os_ts<T> *get();
```

This function performs allocation only the first time it is called in a given process, so you do not need to worry about freeing memory associated with typespecs.

Typespecs for Fundamental and Pointer Types

You can retrieve a typespec for a fundamental or pointer type with one of the following members of the class `os_typespec`:

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

Example

```
unsigned int len ;
os_database* db1 ;
...
char *s = new( db1, os_typespec::get_char(), len ) char[len];
```

os_typespec::get_name()

```
virtual const char* get_name() const = 0;
```

Returns the name of the type designated by the specified typespec. This function does not perform any allocation.

os_typespec::get_typespec()

```
static os_typespec* get_typespec(const char* name);
```

Returns the typespec with the specified name. Returns `0` if the typespec is not found.

os_typespec::get_size()

```
virtual os_unsigned_int32 get_size() const = 0;
```

Returns the size in bytes of instances of the type designated by the specified typespec.

os_typespec::needs_relocation()

```
virtual os_boolean needs_relocation() const = 0;
```

Returns nonzero if the type designated by the specified typespec requires relocation because it contains pointers; returns `0` otherwise.

Retrieving the Database Containing a Specified Object

You can retrieve a pointer to the database that contains a specified object with `os_database::of()`. The function also indicates whether a specified object is transient. The function signature is

```
static os_database* of(const void* obj) ;
```

This function returns a pointer to the database that contains the specified object. If the specified object is not stored in a persistent database, PSE Pro returns a pointer to the transient database that contains the object.

See also `objectstore::is_persistent()`.

Example

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

This example shows how to store two objects together, either both in the same database or both in transient memory. When an instance of `employee` is created, the character array pointed to by `employee::name` is stored together with the new employee.

Retrieving the Transient Database

A transient database represents nondatabase memory, that is, regular program heap memory. When you use persistent new, if you pass a pointer to a transient database, PSE Pro allocates the new object on the transient heap.

`os_database::of()` returns a pointer to a transient database when executed on a transient object. You can also retrieve a pointer to a transient database with `os_database::get_transient_database()`. The function signature is

```
static os_database* get_transient_database() ;
```

This function returns a pointer to a transient database.

Determining Whether an Object Is Persistent

To determine whether an object is persistently stored in a database, call `objectstore::is_persistent()`. The function signature is

```
static os_boolean is_persistent(const void*);
```

This function returns nonzero (true) if the specified address points to persistent memory, and returns `0` (false) otherwise.

Establishing Roots as Database Entry Points

A *database root* provides a way to give an object a persistent name. This allows the object to serve as an *entry point* (or starting point to access other objects in the database) into a database. To help you use database roots, this section covers the following:

- About Database Roots on page 54
- Creating Database Roots on page 55
- Setting the Persistent Object Pointed To by a Root on page 55
- Obtaining the Name of a Root on page 55
- Example of Establishing an Entry Point on page 56

About Database Roots

When an object has a persistent name, any process can look it up by that name to retrieve it. After you have retrieved one object, you can retrieve any object related to it by using navigation. That is, you can retrieve additional objects by following data member pointers. Each database needs only one or a small number of entry points. Looking up an entry point is significantly slower than navigating to an object.

A database root, an instance of `os_database_root`, associates an object with a string. You specify the name of the object when you create a root, with `os_database::create_root()`. You set and get a pointer to the entry-point object with `os_database_root::set_value()` and `os_database_root::get_value()`.

An object can be used as an entry point if you associate a string with it using a database root, an instance of the class `os_database_root`. To do this, use these two functions:

- `os_database::create_root()`
- `os_database_root::set_value()`

Creating Database Roots

Call the `os_database::create_root()` function to create a database root. The function signature is

```
os_database_root* create_root(const char* name);
```

Creates a root named *`name`* in this database, and returns a pointer to the new root. The function copies the specified string into the database; therefore, *`name`* can be a transient string. The root forms an association between the name and an as-yet-unspecified entry-point object. You specify the entry point with `os_database_root::set_value()`. An example appears later in this section.

Each root's sole purpose is to associate an object with a name.

If *`name`* is already associated with a root in the specified database, PSE Pro throws `os_err_root_exists`.

If `this` is a transient database, PSE Pro throws `os_err_database_not_open`.

Setting the Persistent Object Pointed To by a Root

Call the `os_database_root::set_value()` function to associate a root with an object in the database. The function signature is

```
void set_value(const void* value, os_typespec* typespec = 0);
```

This function establishes *`value`* as the value of the specified root. *`value`* is a pointer to an entry-point object. An example of establishing entry points appears later in this section.

`value` must point to persistent memory in the database that contains the root. If *`value`* points to transient memory or memory in another database, PSE Pro throws `os_err_invalid_root_value`.

`typespec` is an optional argument used for type safety.

Obtaining the Name of a Root

Call the `os_database_root::get_name()` function to obtain the name of a particular root. The function signature is

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

This function returns the string associated with the specified root.

Example of Establishing an Entry Point

This example shows how to associate an object with a name using a root, so the object can be used as an entry point.

```
os_database* db1;
image_map *an_image_map;
...
an_image_map =
  new( db1, os_ts<image_map>::get() ) image_map(111);
os_database_root *a_root = db1->create_root("image_map_0");
a_root->set_value(an_image_map);
```

In this example, a root is created with the `os_database` class function `os_database::create_root()`. The function returns a pointer to an instance of the class `os_database_root`. The `this` argument for the function must be the database in which the entry point is stored.

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

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

Retrieving Roots for Database Entry

After one process creates a root and sets its value to point to an entry point, other processes (or the same process) can retrieve the entry point with the following functions:

- `os_database::find_root()`
- `os_database_root::get_value()`
- `os_database::get_all_roots()`

An example appears at the end of this section.

Obtaining a Pointer to a Root

Call the `os_database_root::find_root()` function to obtain a pointer to a root. The function signature is

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

This function returns a pointer to the root with the specified name in the `this` database. This function returns `0` if no root with the specified name is found.

Obtaining a Pointer to an Entry-Point Object

Call the `os_database_root::get_value()` function to obtain a pointer to an entry-point object. The function signature is

```
void* get_value(os_typespec* typespec = 0) const;
```

This function returns the value of the specified root. A root's value is a pointer to an entry-point object. The return value is typed as `void*` so a cast is typically necessary when you use this function.

`typespec` is an optional argument used for type safety.

Obtaining Access to All Roots in a Database

Call the `os_database::get_all_roots()` function to obtain access to all roots in a database.

```
void get_all_roots(
  os_unsigned_int32 max_to_return,
  os_database_root ** roots,
  os_unsigned_int32& n_ret) const;
);
```

The value you specify for *`max_to_return`* is the maximum number of roots to be returned. It also indicates the number of elements contained in `os_database_root`, which is an array of pointers to roots. You must allocate this array before you call `get_all_roots()`. The argument *`n_ret`* specifies the actual number of elements in the array.

Use the function `os_database::get_n_roots()` to determine how large an array to allocate.

```
os_unsigned_int32 get_n_roots() const;
```

This function returns the number of roots in the database.

Example of Retrieving Entry Points

```
os_database *db1 ;
image_map *an_image_map ;
os_database_root *a_root ;
...
a_root = db1->find_root("image_map_0") ;
if (a_root)
  an_image_map = (image_map*) (a_root->get_value()) ;
```

The `this` argument of `os_database::find_root()` (`db1` in this example) is a pointer to the database that contains the root you want to look up. The other argument ("`image_map_0`" in this example) is the root's name. If a root with that name exists in the specified database, PSE Pro returns a pointer to it. If there is no such root, the function returns `0`.

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

Type Safety for Database Roots

You can gain some additional type safety in your use of roots by supplying an `os_typespec*` as the last argument to `os_database_root::set_value()` and `os_database_root::get_value()`.

The typespec should designate the type of the entry-point object, which is the object pointed to by the root's value. The first function stores the typespec in the database. The second function throws `os_err_root_type_mismatch` if the specified typespec does not match the stored one.

Example of Specifying a Typespec for a Root

```
os_database *db1;
image_map *an_image_map;
...
an_image_map = new( db1, os_ts<image_map>::get() ) image_map(111);
db1->create_root("image_map_0")->set_value(
  an_image_map,
  os_ts<image_map>::get()
);
```

In this example, the `os_typespec*` argument to `set_value()` and `get_value()` points to a typespec for `image_map` (not `image_map*`). The `get_value()` function checks only that the typespec supplied to it matches the stored typespec, and does not check the type of the entry-point object itself.

Obtaining the Typespec for a Root

Call the `os_database_root::get_typespec()` function to obtain the typespec for a root. The function signature is

```
os_typespec* get_typespec() const;
```

This function returns the `os_typespec` object last passed to `os_database_root::set_value()` for the root.

Deleting Database Roots

If you want to give an entry point object a different name, or stop using it as an entry point, you can delete its associated root. Call the `os_database_root::destroy()` function to delete a root. The function signature is

```
static void destroy(os_database_root *r);
```

When you destroy a root, this function deletes the associated persistent string as well, but the associated entry-point object is not deleted. If `r` is `0`, the call has no effect. Here is an example of deleting a root:

```
os_database_root *a_root;
...
if (a_root)
  os_database_root::destroy(*a_root);
```

Caution

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

Chapter 5 Memory and Database Management Tasks

This chapter provides information about a variety of tasks you can perform with PSE Pro. It presents information on the following topics:

Establishing Fault Handlers

PSE Pro must handle all memory access violations, because some are actually references to persistent memory in a database. On Windows, there is no function for registering a signal handler for such access violations that also works when you are debugging a PSE Pro application. (The `SetUnhandledExceptionFilter()` function does not work when you use the Visual C++ debugger.)

Every PSE Pro application must put a handler for access violations at the top of every stack in the program. This normally means putting a handler in a program's `main()` or `WinMain()` function and, if the program uses multiple threads, putting a handler in the first function of each new thread.

Use the following macros to establish a handler:

- `OS_PSE_ESTABLISH_FAULT_HANDLER` establishes the start of the fault handler block.
- `OS_PSE_END_FAULT_HANDLER` ends the fault handler block.

Example

```
int main (int argc, char** argv) {
  OS_PSE_ESTABLISH_FAULT_HANDLER
    ...your code...
  OS_PSE_END_FAULT_HANDLER
  return value;
}
```

Braces and semicolons are not necessary because they are part of the macros.

Setting PSE Pro Behavior Before Throwing Exceptions

You can set the `OS_PSE_DEF_EXCEPT_ACTION` environment variable to specify an action that you want PSE Pro to perform just before it throws an exception.

On UNIX, setting this environment variable makes it easier to debug unhandled exceptions. The value you specify determines whether PSE Pro returns control to the debugger, dumps core, or exits with a particular return value. For details, see OS_ PSE_DEF_EXCEPT_ACTION on page 76.

Using Fetch Hooks

An object is retrieved from its database when the page containing it is first needed in a session (a session is the period between database open and database close). So each object is retrieved at most once per session (unless you explicitly reclaim swap space). For any class, you can register a function that PSE Pro calls whenever it retrieves an instance of the class from a database. Such a function is called a *fetch hook* or *hook function.*

To help you use fetch hooks, this section discusses

- Advantages of Fetch Hooks on page 64
- Registering Hook Functions on page 64
- Unregistering Hook Functions on page 65
- Coding Hook Functions on page 65
- Example of Using a Hook Function on page 65
- Setting the Value of a Pointer from a Hook Function on page 66

Advantages of Fetch Hooks

You can use a fetch hook to

- Set pointer-valued data members of the class to the appropriate type of null value. Do this for data members that point to transient (nonpersistent) objects, such as graphical user interface (GUI) objects like windows and dialog boxes. That way the members are null when the object is first retrieved, and the application can subsequently set the members to the appropriate transient addresses. See Example of Using a Hook Function on page 65.
- Set pointer-valued fields of union-valued data members. If a class in your application has a union-valued data member with a pointer field, you should register a fetch hook for that class. The hook function should determine if the active field is a pointer field, and if it is, the function should set the field's value using the PSE Pro macro `OS_REL_PTR()`. See Setting the Value of a Pointer from a Hook Function on page 66.

Registering Hook Functions

You register a hook function for a class, `C`, with `os_ts<C>::set_fetch_hook()`. This function, for any instantiation of the class template `os_ts<T>`, is inherited from the following member of `os_typespec`:

```
typedef void (*os_obj_fetch_hook)(
  void* object,
  os_int32 _os_o,
  void* user_data,
  void* reserved
);
```

```
static void set_fetch_hook(
  os_int32 hook_type,
  os_obj_fetch_hook hook,
  void* user_data
);
```

`hook_type` is a value from the following enumeration, defined in the scope of `os_typespec`:

```
enum { before_hook, after_hook };
```

For the purposes described here, use `os_typespec::after_hook`. If you specify `os_typespec::before_hook`, the function is called *before* PSE Pro adjusts any pointers in the object being retrieved. If you specify `os_typespec::after_hook`, the function is called *after* PSE Pro adjusts any pointers in the object.

`hook` is a pointer to the hook function.

Whenever PSE Pro calls the hook function, it passes *`user_data`* as an argument.

Example

```
os_ts<Window>::set_fetch_hook(
  os_typespec::after_hook,
  Window::fetch_after,
  &count
);
```

Unregistering Hook Functions

You can unregister a hook function for a class, `C`, with `os_ts<C>::unset_fetch_hook()`. This function, for any instantiation of the class template `os_ts<T>`, is inherited from the following member of `os_typespec`:

```
static void unset_fetch_hook(os_int32 hook_type);
```

Coding Hook Functions

Hook functions have the following signature:

```
void function-name(
  void* object,
  os_int32 _os_o,
  void* user_data,
  void* reserved
);
```

`object` is the object being retrieved.

`_os_o` is the offset PSE Pro uses to adjust pointers in the database containing *`object`*. You do not normally need to use this argument (although you must declare it with the name *`_os_o`* if you use `OS_REL_PTR()`).

`user_data` is the pointer specified when the hook function was registered.

`reserved` is reserved for use by a future version of PSE Pro. At present, you do not specify anything.

Example of Using a Hook Function

Suppose instances of the class `Item` are arranged in lists; each instance contains a pointer to another item as the value of the data member `Item::next`. For an instance that terminates a list, `Item::next` is a pointer to a special transient object, the `nil_item`.

If a persistent object contains a pointer to a transient object, that pointer is rendered invalid whenever you close the persistent object's database. Therefore, you must use a fetch hook to reset the pointer each time the object containing it is retrieved.

The hook function `Item::fetch_after()` sets all transient `next` pointers to the transient null object `nil_item`:

```
static Item* nil_item = new Item();

class Item {
  Item *next; // linked list, terminated by transient nil
  static void fetch_after(void*, os_int32, void*, void*);
  ...
};

void Item::fetch_after(void* object, os_int32, void* data, void*)
{
  Item *f = (Item*)object;
  Item *n = f->next;
  if (f->next && !objectstore::is_persistent(f->next)) {
    f->next = nil_item;
  (*(int*)data)++; // increment counter
}
...
int count = 0;
os_ts&lt;Item&gt;::set_fetch_hook(
  os_typespec::after_hook,
  Item::fetch_after,
  &count
);
```

Setting the Value of a Pointer from a Hook Function

For union-valued data members, you can set the value of a pointer field from within a hook function by calling the macro `OS_REL_PTR()`. The hook function's second argument *must* be named `_os_o`.

Enter the macro arguments *without white space*. This macro sets the field value by adding an offset to the pointer. This is necessary to support multidatabase access.

Calls have the form

`OS_REL_PTR(`*`type,member_value`*`)`

`type` is the type pointed to.

`member_value` is the value in the field being set.

Example

```
class Foo {
public:
  union {
    Foo* ptr;
    int i;
  } um;

  enum { is_ptr, is_int };
  int discriminant;

  static void fetch_after(void*, os_int32, void*, void*)
  ...
};

void Foo::fetch_after(void* object, os_int32 _os_o, void*, void*)
{
  Foo *f = (Foo*)object;
  if (f->discriminant == Foo::is_ptr)
  OS_REL_PTR(Foo,f->um.ptr); // requires "_os_o" as offset!
}
```

Troubleshooting Recursive Exceptions

A recursive exception is a memory fault that occurs while processing a memory fault. The most common cause is a fetch hook that accesses unmapped, persistent data. A fetch hook is called while fetching a page after an application has tried to access cold data that has not been mapped from the database into memory during the fault handler. The exception you might see is

```
ObjectStore unhandled exception:
os_err_recursive_fault_detected - Recursive exception detected at
address 0x303d4020.
```

Hashing Persistent Addresses

Use the `objectstore::compute_ptr_hash()` function to obtain a number for use as a hash value. The function signature is

```
static os_unsigned_int32 compute_ptr_hash(const void* addr);
```

This function returns a number suitable for use as a hash value based on *`addr`*, the address of a persistent object. Do not hash on the address directly, since it is not stable from one session to the next.

Reclaiming Swap Space

When PSE Pro transfers data to program memory, it maps the data into the persistent storage region (PSR). This is the range of virtual memory addresses reserved for the database's data. If a large portion of the persistent storage region is used, a large portion of swap space is consumed.

You can free swap space with the following functions:

- `objectstore::return_all_pages()`
- `os_database::return_all_pages()`
- `objectstore::return_memory()`

Removing Data from the Persistent Storage Region

Call the `objectstore::return_all_pages()` function to remove data from the persistent storage region. The function signature is

```
os_unsigned_int32 return_all_pages();
```

All pages are removed *except* the following:

- Pages containing data used only internally by PSE Pro
- Pages updated since the most recent save or commit

If a lot of pages have been updated since the most recent save or commit, consider performing a save or commit prior to calling `return_all_pages()`.

When a page is removed from swap space, all its data still resides in the database and can be retrieved again when needed.

You can call the static function `os_database::return_all_pages()` to perform the same job. The function signature is

```
static os_unsigned_int32 return_all_pages();
```

If your application calls this function on the transient database, PSE Pro throws `os_err_database_transient`.

Removing Data from Virtual Memory

Call the `objectstore::return_memory()` function to remove data from a specified range of virtual memory. The function signature is

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

`address` specifies the beginning of a range of addresses to be removed. The value can be decimal, hexadecimal, or octal.

`length` is an unsigned 32 bit integer specifying the length of the range of memory to be freed. The value can be decimal, hexadecimal, or octal.

`evict_now` specifies whether or not data is removed immediately. If *`evict_now`* is nonzero (true), the data is removed immediately. If *`evict_now`* is 0 (false), the data is not immediately removed, but it is given the highest priority for removal.

All pages in the specified range are removed *except* the following:

- Pages containing data used only internally by PSE Pro
- Pages written since the most recent save or commit

This function cannot remove transient data from virtual memory.

Retrieving Product Information

There is a utility and several functions of the `objectstore` class that return information about the product you are using.

ospsever.exe

Prints the product type and release number of the current installation to `stdout`.

objectstore::which_product()

```
enum os_product_type { pse_pro, ostore_cpp };
static os_product_type which_product();
```

Returns an enumerator indicating which product is used by the current application.

objectstore::release_major()

```
static os_unsigned_int32 release_major();
```

Returns the number preceding the first dot (.) in the number of the release of the product in use by the current application. For example, for Release 4.0, this function returns 4.

objectstore::release_minor()

```
static os_unsigned_int32 release_minor();
```

Returns the number following the first dot (.) in the number of the release of the product in use by the current application. For example, for Release 6.3, this function returns `3`.

objectstore::release_maintenance()

```
static os_unsigned_int32 release_maintenance();
```

Returns the number following the second dot (.) in the number of the release of the product in use by the current application. For example, for Release 6.2.1, this function returns `1`.

objectstore::release_name()

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

Returns the release (major, minor, and maintenance) in string form, for example, `"6.3.0"`.

objectstore::product_name()

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

Returns the product type and release name of the current application in string form, for example, `"PSE Pro for C++ 6.3"`.

Using PSE Pro and ObjectStore for C++ in the Same Application

Although you cannot access a PSE Pro database from ObjectStore, you can use PSE Pro and ObjectStore in the same application to migrate data to an ObjectStore database. The following sections provide information about

- Including Header Files on page 69
- Establishing Fault Handlers on page 70
- Specifying Namespaces on page 70
- Setting Up Address Space on page 70
- Deleting Objects on page 71
- Registering Transient Delete Functions on page 71

See also Example of Using PSE Pro and ObjectStore on page 71.

Including Header Files

Include all ObjectStore headers *before* the PSE Pro header file:

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

Establishing Fault Handlers

The ObjectStore fault handler macros must be *outside* the PSE Pro fault handler macros:

```
OS_ESTABLISH_FAULT_HANDLER;
  OS_PSE_ESTABLISH_FAULT_HANDLER;
    ...
  OS_PSE_END_FAULT_HANDLER;
OS_END_FAULT_HANDLER;
```

Specifying Namespaces

Prefix all PSE Pro class names with `os_pse::`. For example:

```
os_pse::os_database* db = 0;
db = os_pse::os_database::create("a.db",0640,1);
```

The classes of the PSE Pro API are defined within the C++ namespace `os_pse`. If the ObjectStore header is *not* included, the prefix is unnecessary. If the ObjectStore header *is* included (prior to the PSE Pro header), the prefix is required for *all* PSE Pro classes.

Setting Up Address Space

Use one of the following mechanisms to set up disjoint address space ranges for ObjectStore and PSE Pro:

- Perform *full* ObjectStore initialization *before* initializing PSE Pro. Full ObjectStore initialization is performed during the first ObjectStore transaction. For example:

  ```
  objectstore::initialize();
  OS_BEGIN_TXN(fullinit,0,os_transaction::read_only)
  {
    ; // full initialization
  }
  OS_END_TXN(fullinit);
  ```

  ```
  os_pse::objectstore::initialize();
  ```

- Set the start of the PSE Pro persistent storage region (PSR) to a value that does not interfere with the ObjectStore PSR. The default PSR for both ObjectStore and PSE Pro starts at 0x3000000 and has a size of 0x08000000 (128 MB).

  For example, set the start of the PSR for PSE Pro to 0x8000000:

  ```
  set OS_PSE_AS_START=134217728
  ```

Deleting Objects

If ObjectStore and PSE Pro are used in the same application, PSE Pro defines `operator::delete()` with the following `inline` function (see `os_pse/api/os_pnew.hh`):

```
inline void operator delete(void* p)
{
  if (objectstore::is_persistent(p))
    objectstore_delete(p);
  else
    os_pse::objectstore::free_memory(p);
}
```

If the object to be deleted is not in the ObjectStore PSR, the PSE Pro `free_memory()` function calls the PSE Pro transient `delete` function if there is one, or calls `free()` if there is no PSE Pro transient `delete` function.

Registering Transient Delete Functions

If the application uses its own transient delete function, the function should be registered with both PSE Pro and ObjectStore:

```
void del_func(os_void_p p) {
  free(p);
}
objectstore::set_transient_delete_function(del_func);
os_pse::objectstore::set_transient_delete_function(del_func);
```

Example of Using PSE Pro and ObjectStore

This is a simple application for copying data from a PSE Pro database to an ObjectStore database.

```
/*** file foo.hh ***/

class Foo {
public:
  ~Foo();
  Foo(int k) { i=k; }
   int i;
};

Foo::~Foo() {
  char *storage = "trans";
  if (objectstore::is_persistent(this))
  storage = "OStore";
  else if (os_pse::objectstore::is_persistent(this))
  storage = "PSE";
  printf("Foo %d dtor(0x%x) [%s] called.\n", i, this, storage);
}

/*** file test.cpp ***/

#include <ostore/ostore.hh>
#include <os_pse/ostore.hh>
#include <stdlib.h>
#include "foo.hh"

void del_func(os_void_p p)
{
  // fprintf(stdout,"del_func(0x%x)\n", p);
  // do not call delete here because doing so causes an infinite
  //recursion
  free(p);
}
int main(int argc, char* argv[])
{
  char *str = new char[10];
  delete[] str;

  OS_ESTABLISH_FAULT_HANDLER;
  OS_PSE_ESTABLISH_FAULT_HANDLER;

  objectstore::initialize();
  objectstore::set_transient_delete_function(del_func);

  OS_BEGIN_TXN(fullinit,0,os_transaction::read_only) {
  ; // force full init, or set OS_PSE_AS_START
  } OS_END_TXN(fullinit);

  os_pse::objectstore::initialize();
  os_pse::objectstore::set_transient_delete_function(del_func);
  os_pse::os_transaction::initialize();

  os_database* os_db = 0;
  os_pse::os_database* pse_db = 0;

  os_db = os_database::create("c:\\temp\\os.db",0640,1);
  pse_db = os_pse::os_database::create(
    "c:\\temp\\pse.db",0640,1);

  os_transaction *os_txn =
  os_transaction::begin(os_transaction::update);
  os_pse::os_transaction *pse_txn =
  os_pse::os_transaction::begin(
    os_pse::os_transaction::update);
  {
  os_db->create_root("str");
  pse_db->create_root("str");

  char * os_str = new(os_db, os_typespec::get_char(), 10) char[10];
  char * pse_str = new(
    pse_db, os_pse::os_typespec::get_char(), 10) char[10];

  Foo* foo = new Foo(1);
  delete foo;

  foo = new(pse_db, os_ts<Foo>::get()) Foo(2);
  delete foo;

  foo = new(os_db, os_ts<Foo>::get()) Foo(3);
  delete foo;

  }
  os_transaction::commit();
  os_pse::os_transaction::commit();

  pse_db->save();
  pse_db->close();
  os_db->close();

  OS_PSE_END_FAULT_HANDLER;
  OS_END_FAULT_HANDLER;

  return 0;
```

```
}

/*** file schema.cpp ***/

#include <ostore/ostore.hh>
#include <ostore/coll.hh>
#include <ostore/manschem.hh>
#include "foo.hh"

void dummy ()
{
  OS_MARK_SCHEMA_TYPE(Foo);
}
```

Here is the application's makefile:

```
INCLUDES = /Ic:\msdev\include \
/I$(OS_ROOTDIR)/include /I$(OBJ_TOP_DIR)\include
DEFS = /DREGISTER=register -DWIN32
CCC = cl
CCC_DEBUG = /Zi
CCC_FLAGS = /GX /MD $(DEFS) $(INCLUDES) \
$(CCC_DEBUG) /nologo
LINKOPTS= /nologo /debug /debugtype:cv -map:$*.map
LIBS=$(OBJ_TOP_DIR)\lib\osclt.lib $(OS_ROOTDIR)/lib/ostore.lib

SCHEMA_SRC= schema.cpp
APP_SCHEMA_SRC= osschema.cpp
APP_SCHEMA_OBJ= osschema.obj
APP_SCHEMA_DB= c:\temp\testpse.adb
LIB_SCHEMA_DBS=

PROGRAMS= test.exe

.SUFFIXES:
.SUFFIXES: .obj .cpp .ii
.cpp.obj:
  $(CCC) $(CCC_FLAGS) /c /Fo$@ /Tp$<
.cpp.ii:
  $(CCC) /E $(CCC_FLAGS) /Tp$< > $@

all: $(PROGRAMS)

test.exe: test.obj $(APP_SCHEMA_OBJ)
  $(CCC) /Fetest.exe test.obj $(APP_SCHEMA_OBJ) $(LIBS) \
/link $(LINKOPTS)

# schema stuff
$(APP_SCHEMA_OBJ): $(SCHEMA_SRC)
  ossg -asof $(APP_SCHEMA_OBJ) \
-asdb $(APP_SCHEMA_DB) $(CPPFLAGS) $(SCHEMA_SRC)

clean:
  del *.obj *.exe *.pdb *.map *.db \
$(APP_SCHEMA_DB) $(APP_SCHEMA_SRC)
```

Prefaulting Persistent Data

ObjectStore and PSE Pro use a faulting mechanism to fetch persistent data from disk and to detect write access.

Some libraries, for example NT system libraries, do not allow pointers to unmapped or protected data to be passed to functions. In this case, an object can be manually faulted in with the `objectstore::touch()` function. The function signature is

```
static void touch(const void* obj, os_boolean for_write);
```

If ***`obj`*** points to a persistent object that is not mapped into memory, `touch()` fetches it from the database and maps it into memory. If ***`for_write`*** is true, the data is mapped with read/write permissions. If the object pointed at by ***`obj`*** is already mapped, `touch()` has no effect.

Units of transfer in PSE Pro are memory pages (4096 bytes). A single touch on a particular object maps the entire page the object is stored in.

Using Integer Typedefs

The PSE Pro API uses the following typedefs for integer types:

```
typedef int os_int32;
typedef unsigned int os_unsigned_int32;
typedef int os_boolean;
```

PSE Pro Environment Variables

PSE Pro includes the following environment variables:

- `OS_PSE_AS_SIZE` on page 75
- `OS_PSE_AS_START` on page 75
- `OS_PSE_CHK_ILLEGAL_PTR` on page 75
- `OS_PSE_DEF_BREAK_ACTION` on page 75
- `OS_PSE_DEF_EXCEPT_ACTION` on page 76
- `OS_PSE_PAGE_SIZE` on page 76
- `OS_PSE_TRACE_ALLOC` on page 78
- `OS_PSE_VERIFY_ALLOC` on page 78

OS_PSE_AS_SIZE

This environment variable specifies the size, in bytes, of your application's persistent storage region. This is the range of virtual memory addresses reserved for database data. Specify the number of bytes in decimal notation. The default is 128 MB (0x08000000). You do not normally need to change the default.

An incorrect value for `OS_PSE_AS_SIZE` can cause failures. Be absolutely certain you understand how addresses are assigned on your platform before you modify this value.

On Windows, two gigabytes is the maximum user address space. Your program, its data, and the persistent storage region must fit in two gigabytes. A user program on Windows uses the bottom two gigabytes of address space. DLLs use up space at 0x10000000; stacks use up space at 0x7fxxxxxx. Consequently, there are only 1.75 gigabytes available.

If necessary, PSE Pro rounds down the number you specify to a page-size boundary, that is, a multiple of the page size.

OS_PSE_AS_START

This environment variable specifies the starting location of an application's persistent storage region. Specify the location in decimal notation. The default is 0x30000000. You do not normally need to change the default.

An incorrect value for `OS_PSE_AS_START` can cause failures. Be absolutely certain you understand how addresses are assigned on your platform before you modify this value.

If necessary, PSE Pro rounds down the number you specify to a page-size boundary, that is, a multiple of the page size.

OS_PSE_CHK_ILLEGAL_PTR

Indicates whether you want PSE Pro to check for invalid pointers in a page after relocation. When set, PSE Pro checks all pointers in relocated pages, displays a message that indicates invalid pointers, and then continues processing.

The default is that this variable is not set.

OS_PSE_DEF_BREAK_ACTION

Setting this environment variable enables just-in-time debugging. With this feature enabled, an uncaught exception thrown by PSE Pro causes a dialog box to be displayed, giving you the option of entering the debugger, with the stack intact.

OS_PSE_DEF_EXCEPT_ACTION

Setting this environment variable specifies an action that you want PSE Pro to perform when it is about to throw an exception.

On UNIX, setting this environment variable makes it easier to debug unhandled exceptions. The value you specify determines whether PSE Pro returns control to the debugger, dumps core, or exits with a particular return value.

By default, this variable is not set. When PSE Pro encounters an exception and this variable is not set, PSE Pro throws the exception to the program. If the program does not catch and handle the exception, the PSE Pro fault handler macros (`OS_PSE_ESTABLISH_FAULT_HANDLER` and `OS_PSE_END_FAULT_HANDLER`) print a message and then terminate the program.

You can specify the following values to set this variable:

Value	*Description*
`abort`	PSE Pro aborts the process. On UNIX, PSE Pro creates a core file and, if you are running in a debugger, returns control to the debugger.
`integer`	Specify an integer greater than or equal to 1. PSE Pro exits the program with the specified integer as the return value.
`kill`	PSE Pro action varies by platform: • Windows — `ExitProcess (0x006600);` • UNIX — `kill (getpid(), SIGKILL);`
Any other value	PSE Pro exits the program with a return value of `1`. This is the default value for setting this variable.

OS_PSE_PAGE_SIZE

This feature supports a variable soft page size for PSE Pro. It enables the user to set the soft page size value. Setting a higher page size than the default 4K generally increases performance and increases the address range that can be mapped to values greater than normally specified by operating system limitations.

To use this feature, set the environment variable OS_PSE_PAGE_SIZE to the value in kilobytes of the page size. To set a page size of 32K use

```
# setenv OS_PSE_PAGE_SIZE 32
```

The variable page size settings are restricted by session and database. This means that

- An application cannot use two different page size settings during a session.
- The page size under which a database is created is embedded into the database; trying to use it with a different page size setting will cause an exception to be thrown.

To support page sizes large than 4K, PSE's internal data structures used to manage objects and pages are modified in order to minimize overhead and constraint. The tables below illustrate the size variations of the object header (per object) and the page header (per page) under different page size settings.

Object Header Size

These sizes apply to all supported platforms.

Page Size	*Object Header Size*	*Number of Types Supported Per DB*
4K	8 Bytes	16K
8K	8 Bytes	4K
16K	8 Bytes	4K
32K	8 Bytes	4K
64K	16 Bytes	16K
128K	16 Bytes	16K
256K and up	16 Bytes	16K

Page Header Size

Page header sizes vary by platform and page size as follows:

Platform	*Page Header Size (page size < 64K)*	*Page Header Size (page size >= 64K)*
Windows XP/2000	24 Bytes	32 Bytes
HP-UX 11.0 (32 bit)	24 Bytes	32 Bytes
HP-UX 11.00 (64 bit)	40 Bytes	40 Bytes
Linux (Redhat 6.2)	32 Bytes	32 Bytes
Solaris (Forte CC Sun WorkShop 6.1)	24 Bytes	32 Bytes

OS_PSE_TRACE_ALLOC

When this environment variable is set, output is generated that describes every allocation and deallocation of persistent memory.

OS_PSE_VERIFY_ALLOC

When this environment variable is set, each time persistent memory is allocated or deallocated, PSE Pro checks the integrity of the free list and allocated list data structures for the page on which the allocation or deallocation occurred. If PSE Pro detects a problem, it throws `os_err_heap_corruption`. The corruption can be due to a memory corruption problem in user code or a product defect.

Chapter 6 Transactions

With PSE Pro, you can include transactions in your applications. To help you use transactions, this chapter covers

Introducing Transactions

A transaction is a logical unit of work. It is a consistent and reliable portion of the execution of a program. In your code, you initialize the transaction facility, and then you place calls to the PSE Pro API to mark the beginnings and ends of transactions. After you initialize the transaction facility, access to persistent objects must always take place inside a transaction.

When a transaction ends, either the database is updated with all of the transaction's changes to persistent objects, or the database is not updated at all. If a failure occurs in the middle of a transaction, or you decide to abort the transaction, the contents of the database remain unchanged.

The transactions API includes the following functions:

- `os_transaction::initialize()` initializes the transaction facility.
- `os_transaction::begin()` begins a transaction.
- `os_transaction::commit()` commits a transaction, which saves all databases.
- `os_transaction::abort()` aborts a transaction, which removes any changes made since the transaction started.

For databases that are operating system files, if a failure occurs during a transaction, use the recovery facility to recover your data as of the transaction's beginning.

For databases that are streams within OLE compound files, use `TRANSACTED` mode.

Nested transactions are not allowed.

Initializing the Transaction Facility

Call the `os_transaction::initialize()` function to initialize the transaction facility. The function signature is

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

You must call this function before using transactions. Call this function at most once per run of an application, before opening any databases. If you open a database and then call `initialize()`, PSE Pro throws `os_err_trans`.

After the call is made, all access to database data must take place within a transaction. (Database create and close operations should take place outside transactions.) If you attempt to access persistent data outside a transaction, PSE Pro throws `os_err_no_trans`.

Beginning Transactions

Call the `os_transaction::begin()` function to start a transaction.

```
static os_transaction* begin(transaction_type_enum type = update);
```

```
static os_transaction* begin(
  const char* name,
  transaction_type_enum type = update
);
```

Both overloadings return a pointer to the new transaction object.

Specifying **transaction_type_enum**

You can specify `os_transaction::transaction_type_enum()` as follows:

```
enum transaction_type_enum { none, update, read_only };
```

The enumerators `os_transaction::update` and `os_transaction::read_only` can be arguments to `os_transaction::begin()`. PSE Pro also uses them as return values for `os_transaction::get_type()`. When there is no transaction, `os_transaction::none` is the return value for `os_transaction::get_type()`.

Specifying Other Parameters

`name` specifies an identifier for the new transaction, which you can retrieve with `os_transaction::get_name()`.

If *`type`* is `os_transaction::read_only`, attempts to write persistent data within the transaction result in `os_err_trans_wrong_type`. See `os_transaction::get_type()`.

Exceptions

If you call this function inside a transaction, PSE Pro throws `os_err_trans`.

If the transaction facility has not been initialized, PSE Pro throws `os_err_trans`.

Setting the Name of a Transaction

Call `os_transaction::set_name()` to give a transaction a name. The function signature is

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

This function sets the name of the specified transaction. The function copies the string *`new_name`*. See also `os_transaction::begin()` and `os_transaction::get_name()`.

Obtaining a Pointer to the Current Transaction

Call the `os_transaction::get_current()` function to obtain a pointer to the current transaction. The function signature is

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

This function returns a pointer to the current transaction. If there is no transaction in progress, this function returns `0`. See also `os_transaction::begin()`.

Ending Transactions

There are two ways to end a transaction:

- `os_transaction::commit()` makes any changes permanent in the database or databases.
- `os_transaction::abort()` rolls back any changes made since the transaction started.

See Example of Using Transactions on page 83.

Making Changes Permanent

To end a transaction and save your changes, call the `os_transaction::commit()` function. The function signature is

```
static void commit();
```

This function commits the current transaction. All the transaction's changes to persistent data are made permanent in the databases.

If multiple databases are open and a failure occurs during a commit, some databases might contain the transaction's changes while others do not. This release does not perform a two-phase commit. However, any given database involved in the transaction has either all the changes or none of them.

If there is no currently active transaction, PSE Pro throws `os_err_no_trans`.

See also `os_transaction::get_current()` and `os_transaction::is_committed()`.

Rolling Back Changes

To end a transaction and discard any changes made during the transaction, call the `os_transaction::abort()` function. The function signature is

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

This function terminates the current transaction and rolls back database data to its state as of the beginning of the transaction. The abort renders a pointer invalid if it points to persistent memory allocated during the transaction or to a database root created during the transaction.

If there is no current transaction, PSE Pro throws `os_err_no_trans`.

See also `os_transaction::get_current()` and `os_transaction::is_aborted()`.

Obtaining Information About Transactions

You can obtain the following information about a transaction:

- Is This Transaction Committed? on page 82
- Is This Transaction Aborted? on page 82
- What Is This Transaction's Name? on page 82
- What Is This Transaction's Type? on page 83

Is This Transaction Committed?

Call `os_transaction::is_committed()` to determine if the current transaction has been committed:

```
os_boolean is_committed() const;
```

This function returns nonzero if the specified transaction has been committed, and `0` otherwise. See also `os_transaction::commit()`.

Is This Transaction Aborted?

Call `os_transaction::is_aborted()` to determine if the current transaction has been aborted:

```
os_boolean is_aborted() const;
```

This function returns nonzero if the specified transaction has been aborted, and `0` otherwise. See also `os_transaction::abort()`.

What Is This Transaction's Name?

Call `os_transaction::get_name()` to obtain the name of the current transaction:

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

This function returns the name of the specified transaction. It is the caller's responsibility to deallocate the string when it is no longer needed. See also `os_transaction::begin()` and `os_transaction::set_name()`.

What Is This Transaction's Type?

Call `os_transaction::get_type()` to determine whether you can update data during the current transaction:

```
transaction_type_enum get_type() const;
```

This function returns either os_transaction::update or `os_transaction::read_only`, depending on which is true of the current transaction. If there is no current transaction, this function returns os_transaction::none.

Example of Using Transactions

Consider replacing an assembly's subparts and then performing a constraint check. Suppose that if the constraint check fails, you want to undo the replacement.

```
void replace_image(
  image_map *an_image_map,
  image an_image, a_new_image
)
{
  os_transaction::begin();
  an_image_map->children.remove(an_image);
  an_image_map->children.insert(a_new_image);
  if (!check_size(an_image_map)) {
    cout << "change aborted: size check failed\n";
    os_transaction::abort(); /* undo the image replacement */
  }
  else
    os_transaction::commit();
}
```

This example uses the following functions:

- `os_transaction::begin()`
- `os_transaction::commit()`
- `os_transaction::abort()`

Chapter 7 The Undo/Redo Facility

PSE Pro provides an undo/redo facility that allows you to set undo points in your application. These points mark portions of work that can be undone and subsequently redone. To help you use this facility, this chapter discusses

Setting Undo Points

Setting an undo point copies a database, using a copy-on-write implementation. The first time a page is updated following an undo point, PSE Pro stores a before-image of the page in a transient log (the *undo/redo log*). When the database is rolled back to the undo point, the before-images are swapped with their updated counterparts. If a redo occurs, they are swapped back.

Creating an Undo Point

Call `os_database::set_undo_marker()` to create an undo point:

```
void set_undo_marker();
```

This function sets an undo point, which marks the beginning of an operation that can be undone. The `this` database can subsequently be rolled back to its state as of the time of the call to `set_undo_marker()`.

After you call `set_undo_marker()`, you cannot redo any previously undone operation.

If you specify a transient database, PSE Pro throws `os_err_database_transient`.

If you specify a read-only database, PSE Pro throws `os_err_write_permission_denied`.

Setting the Maximum Number of Undo Operations

To gain some control over the maximum size of the undo/redo log, you should set a limit on the number of undo points for which PSE Pro can store before-images. To do this, call `objectstore::set_maximum_undo_operations()`.

```
void objectstore::set_maximum_undo_operations(
  os_unsigned_int32 m
);
```

This function sets the maximum number of marked operations that can be undone. If you specify `0`, or if you do not call this function during the current process, there is no maximum.

Undoing Database Updates

Call `os_database::undo()` to undo changes made to the database:

```
void undo(os_unsigned_int32 n_markers = 1);
```

This function undoes the changes that were made to the specified database during the previous *`n_markers`* marked operations (not counting operations that are already undone).

A marked operation consists of the updates performed to a given database between one undo point and the next or between an undo point and a call to `undo()`. So `undo()` rolls back the specified database to its state as of the undo point marking the beginning of the last *`n_markers`* (not currently undone) marked operations.

Calling `undo()` renders transient state invalid. After calling `undo()`, you should reretrieve transient pointers to persistent objects.

Undo only undoes marked operations performed since the last open, save, or commit of the database. If there are fewer than *`n_markers`* undo points following the last open, save, or commit, the database is rolled back to the first undo point following the last open, save, or commit. If there are no undo points following the last open, save, or commit, `undo()` has no effect.

If the last call to `objectstore::set_maximum_undo_operations()` specified a maximum less than *`n_markers`*, PSE Pro undoes only the maximum number of marked operations.

Redoing Undone Database Updates

Call `os_database::redo()` to redo undone database operations:

```
void redo(os_unsigned_int32 n = 1);
```

This function redoes the *n* most recently undone marked operations.

`redo()` only redoes marked operations undone since the last update to the specified database or call to `os_database::set_undo_marker()` for the database. A marked operation consists of the updates performed to a given database between one undo point and the next or between an undo point and a call to `undo()`.

Calling `redo()` renders transient state invalid. After calling `redo()`, you should reretrieve transient pointers to persistent objects.

If there are fewer than *n* undone marked operations since the last update to the specified database or call to `os_database::set_undo_marker()`, PSE Pro redoes correspondingly fewer operations. If there are no undone marked operations since the last update to the specified database or call to `os_database::set_undo_marker()`, `redo()` has no effect.

Determing If Changes Can Be Undone or Redone

To determine if there is an operation that can be undone, call `os_database::is_undoable()`:

```
os_boolean is_undoable();
```

This function returns nonzero (true) if, subsequent to the last open, save, or commit for the specified database, there is at least one undo point marking the beginning of an operation that is not currently undone; it returns `0` (false) otherwise.

To determine if there is an undone operation that can be redone, call `os_database::is_redoable()`:

```
os_boolean is_redoable();
```

This function returns nonzero (true) if there is at least one currently undone operation subsequent to the last open, save, or commit for the specified database; it returns `0` (false) otherwise.

Determining if a Database Was Modified

To determine if a database was modified after an undo marker was set, call `os_database::is_altered()`:

```
os_boolean os_database::is_altered();
```

Use this function to avoid setting unnecessary undo markers if the database has not been modified. You can call `os_database::is_altered()` to check whether a database has been modified or not. This function returns true if the database was modified after the last call to this function or after an `os_database::set_undo_marker()` is called.

Chapter 8 Object Iteration

PSE Pro provides an API for iterating over objects. This chapter covers

Overview of Object Iteration

An object cursor allows retrieval of the objects stored in a specified database, one object at a time, in an arbitrary order. If no objects are added or deleted from the database, this order is stable across traversals of the database, as long as PSE Pro provides operations for creating a cursor, advancing a cursor, and for testing whether a cursor is currently positioned at an object (or has already returned all objects). It is also possible to position a cursor at a specified object in the database.

In addition, a function is provided for retrieving the object at which a cursor is positioned together with an `os_typespec` representing the type of the object and, for an object that is an array, a number indicating how many elements it has.

The object iteration API includes the following functions:

- `os_object_cursor::current()` retrieves the object.
- `os_object_cursor::first()` positions the cursor at the first object.
- `os_object_cursor::more()` checks the position of a cursor.
- `os_object_cursor::next()` positions the cursor at the next object.
- `os_object_cursor::os_object_cursor()` creates an object cursor.
- `os_object_cursor::~os_object_cursor()` deletes an object cursor.
- `os_object_cursor::set()` positions the cursor.

Creating and Deleting Cursors

PSE Pro provides the `os_object_cursor` class to represent cursors. Here is the constructor for creating a cursor:

```
os_object_cursor(os_database* db);
```

A call to the constructor creates a new `os_object_cursor` associated with the specified database. If the database is empty, the cursor is positioned at no object; otherwise, it is positioned at the first object in the cursor's associated database. The object is first in an arbitrary order that is stable across traversals of the database, as long as no objects are created or deleted from the database.

To delete an object cursor, call the destructor:

```
~os_object_cursor();
```

This function performs internal maintenance associated with `os_object_cursor` deallocation.

Navigating with Cursors

You can iterate through the objects in a database by

- Positioning the Cursor at the First Object on page 90
- Positioning the Cursor at the Next Object on page 90
- Positioning the Cursor at the Specified Object on page 91
- Determining if the Cursor Is Pointing to an Object on page 91

Positioning the Cursor at the First Object

To position the cursor at the first object in the cursor's associated database, call `os_object_cursor::first()`:

```
void first();
```

The object is first in an arbitrary order that is stable across traversals of the database, as long as no objects are created or deleted from the database. If there are no objects in the cursor's associated database, this function has no effect.

Positioning the Cursor at the Next Object

To position the cursor at the next object in the cursor's associated database, call `os_object_cursor::next()`:

```
void next();
```

The object is next in an arbitrary order that is stable across traversals of the database, as long as no objects are created or deleted from the database. If the cursor is currently positioned at no object, `os_err_cursor_at_end` is signaled. Otherwise, this function has no effect.

Positioning the Cursor at the Specified Object

To position the cursor at a specified object, call

```
os_object_cursor::set()
```

```
void set(const void* ptr);
```

Positions the cursor at the object containing the address *ptr*. If *ptr* is not an address in the specified cursor's associated database, `os_err_cursor_not_at_object` is signaled. If *ptr* is in the cursor's associated database but within unallocated space, `os_err_cursor_not_at_object` is signaled.

Determining if the Cursor Is Pointing to an Object

To determine if the cursor is pointing to an object, call `os_object_cursor::more()`:

```
os_boolean more() const;
```

This function returns nonzero (true) if the cursor is positioned at an object. It returns `0` (false) otherwise.

Retrieving the Object at the Cursor Position

If the cursor is positioned at an object, you can retrieve that object with `os_object_cursor::current()`:

```
os_boolean current(
  void* pointer,
  const os_typespec* type,
  os_int32 count) const;
```

If the cursor is positioned at an object, this function returns nonzero (true), sets *pointer* to the address of the object, and sets *type* to an `os_typespec` representing the object's type. If the object is an array, *count* is set to the number of elements it has. If the object is not an array, *count* is set to `0`. If the cursor is not positioned at an object, `0` (false) is returned, and all three arguments are set to `0`.

Example of Using Object Iterators

This example returns the number of characters in a specified database:

```
int count_chars(os_database* db) {
void *ptr=0; os_typespec* ts=0;
os_int32 count=0;
int n_chars = 0;
os_object_cursor c(db);
for (c.first(); c.more(); c.next())
  if(c.current(ptr, ts, count) && (ts == os_ts<ltchar>gt::get()))
    n_chars += count ? count : 1;
return n_chars;
}
```

Chapter 9 Evolving Database Schemas

PSE Pro provides an API and methodology for evolving the schemas of PSE Pro databases. PSE Pro includes a sample schema evolution application. See the README file in `examples/sevol`.

This chapter discusses the following topics:

Introduction to Schema Evolution

Suppose you change the definition of a class or struct that your application uses and you want to continue to use databases that already contain instances with the old definitions. In this situation, your application must have the ability to upgrade existing databases. The process of changing the type definitions known to a PSE Pro database is called *schema evolution*.

The schema evolution API comprises the following functions:

- `os_database::rename_type()`
- `os_database::record_new_object_location()`
- `os_database::fix_pointers()`

Schema evolution is required when you make a change that affects the layout of a class. The result of such changes is that your program can no longer interpret the class instances stored in a PSE Pro database. Incompatible changes to a class definition include the following:

- Adding or removing a data member

- Changing the type of a data member
- Introducing the first virtual function in a class
- Removing the last virtual function from a class
- Adding or removing a base class

Description of the Schema Evolution Process

The schema evolution process consists of the following steps, which are described in detail in the sections that follow:

1. Rename the old class definition and mark both the old and new class definitions as persistent.
2. Add a schema evolution constructor to the new class.
3. Open the database and rename the type of the existing instances in the database, using the function `os_database::rename_type()`.
4. Iterate through the objects in the database using an `os_object_cursor`, and allocate an instance of the new class for each instance of the old class. Copy the state as appropriate and then delete the old instance. See Chapter 8, Object Iteration, on page 89 for information about the `os_object_cursor` class.

   During this iteration, PSE Pro also records the location of each new instance with the function `os_database::record_new_object_location()` as part of the schema evolution constructor.
5. Invoke `os_database::fix_pointers()`.

   This function visits each object in the database and redirects pointers to the old instances so that they now point to the new instances. This function uses the information recorded by `os_database::record_new_object_location()`.
6. Remove the old class and save the database.

   Perform any cleanup operations that might be left over, and call `os_database::save()` to apply the changes of the database permanently.

Step 1: Marking Both Class Definitions Persistent

The first step in implementing schema evolution has two parts:

- Rename the old class definition.
- Mark both the old and the new definitions as persistent in your `schema.scm` file, with the `OS_MARK_SCHEMA_TYPE` macro.

  For the old class, only the data members and base classes are required to be included. If the old class has virtual functions, you should create a single dummy virtual function in the old version of the class.

For example, suppose your application includes the following class:

```
class SevolExample {
private:
  int x;
  SevolExample* y;
public:
  SevolExample(SevolExample* next, int z);
  Int get_x() { return x; }
  SevolExample* get_y() { return y; }
  virtual SomeVirtualFunction();
};
```

Now suppose you want to introduce a new data member. Here is the new class declaration:

```
class SevolExample {
private:
  int x;
  SevolExample* y;
  char* new_data_member; // The new data member.
public:
  SevolExample(SevolExample* next, int z);
  Int get_x() { return x; }
  SevolExample* get_y() { return y; }
  virtual SomeVirtualFunction();
};
```

To upgrade the database, in the `schema.scm` file, mark as persistent a class that has the same definition as the old version of `SevolExample`. Here is the definition of the `old_SevolExample` class, which the same as the definition of the `SevolExample` class:

```
class old_SevolExample {
public:
  int x;
  SevolExample* y;
public:
  virtual dummy() {}
};
```

The `old_SevolExample` class has the identical class layout as the `SevolExample` class before the addition of `new_data_member`. In particular, note that it has a dummy virtual function to maintain the virtual function table. If `SevolExample` had base classes, then `old_SevolExample` would have to have the identical base classes. Note also that all the data members are public. An alternative would be to make `SevolExample` a friend class.

Step 2: Adding a Schema Evolution Constructor to the New Class

Add the following schema evolution constructor to the new `SevolExample` class:

```
class SevolExample {
...
public:
  SevolExample(old_SevolExample* old)
  {
    x = old->x;
    y = old->y;
    new_data_member = 0;
    os_database::of(this)->record_new_object_location(old,this);
    delete old;
  }
  ...
};
```

The schema evolution constructor must do the following:

- Copy the values of data members from the old class to the new class.

  In particular, note that pointer values should be copied blindly. They are fixed up later in the process by `os_database::fix_pointers()`.

- Call `os_database::record_new_object_location()`.

  Passing an `os_database*` argument to the schema evolution constructor is an alternative to calling `os_database::of(this)`. If you know that you are going to delete an object that is pointed to by the current object and you want to make that pointer `NULL`, pass 0xFFFFFFFF as the value of the new object location to `os_database::record_new_object_location()`. If `os_database::fix_pointers()` sees an object address mapped to 0xFFFFFFFF, it sets any pointers to the old address to NULL.

- Delete the old object.

A schema evolution constructor can be much more complicated than the one shown here if the new and old classes are very different. One thing to bear in mind, however, is that the schema evolution constructor should delete only the old object that is passed in. This is because deleting objects while using `os_object_cursor` has to be done very carefully: You can only delete an object *after* the `os_object_cursor` has already visited it. Otherwise, the object cursor can be corrupted. If you need to delete additional objects during schema evolution, it is best to record the objects to be deleted in a separate data structure and then delete them after the schema evolution process is complete.

Step 3: Renaming the Types of the Existing Instances

Having done the preliminary work of marking both the old and new versions of the class as persistent and of supplying a schema evolution constructor, you are now in a position to implement the actual schema evolution process. The first step is to call `os_database::rename_type()` for each type that is to be evolved.

```
void EvolveDatabase(os_database* db)
{
  db->rename_type("SevolExample", "old_SevolExample");
}
```

Steps 4 and 5: Allocating Instances of New Classes and Fixing Pointers

Now, you must iterate through each object in the database, and, for each instance of `old_SevolExample`, allocate an instance of `SevolExample` using the schema evolution constructor:

```
void EvolveDatabase(os_database* db)
{
  db->rename_type("SevolExample", "old_SevolExample");
  os_object_cursor c(db);
  for (c.first(); c.more();) {
    void* ptr;
    os_typespec* ts;
    os_int32 cnt;
    // The example retrieves the current object, and then
    // calls c.next() before allocating the new object.
    c.current(ptr, ts, cnt);
    c.next();
    if (!strcmp(ts->get_name(), "old_SevolExample"))
      new(db, os_ts<SevolExample>::get())
        SevolExample((old_SevolExample*)ptr);
  }
  // Finally, have PSE Pro fix up all the pointers so that
  // pointers to the old objects now point to the new ones
  db->fix_pointers();
}
```

If you have more than one class that has changed, the only changes that you need to make to this function are to add more calls to `os_database::rename_type()` and to add `else if` clauses to the condition that checks the type of the current object and allocates instances of the new class.

```
    if (!strcmp(ts->get_name(), "old_SevolExample"))
      new(db, os_ts<SevolExample>::get())
        SevolExample((old_SevolExample*) ptr);

    else if (!strcmp(ts->get_name(), "old_Class2"))
      new(db, os_ts<Class2>::get())
        Class2((old_Class2*) ptr);
```

Step 6: Removing the Old Class from Your Application

If you decide that your application no longer needs a particular class, schema evolution for that class consists of deleting its instances from the PSE Pro database and setting any pointers to the deleted objects to NULL. If the destructor of the class that is being removed does not delete any other objects, you can handle the operation within the `os_object_cursor` iteration. For example, if you have a class called `NotNeeded` in your database, the following code would remove it:

```
void EvolveDatabase(os_database* db)
{
  db->rename_type("SevolExample", "old_SevolExample");

  os_object_cursor c(db);

  for (c.first(); c.more();) {
    void* ptr;
    os_typespec* ts;
    os_int32 cnt;

    c.current(ptr, ts, cnt);
    c.next();

    if (!strcmp(ts->get_name(), "old_SevolExample"))
      new(db, os_ts<SevolExample>::get())
        SevolExample((old_SevolExample*)ptr);

    else if (!strcmp(ts->get_name(), "NotNeeded") {
      db->record_new_object_location(ptr, 0xffffffff);
      delete (NotNeeded*)ptr;
    }
  }
  db->fix_pointers();
}
```

The operation is a little more complicated if the destructor for the class `NotNeeded` makes calls to operator `delete`. This is because the only object that it is safe to delete while using an `os_object_cursor` is the object that you have just visited. The example retrieves the current object (using `c.current(ptr, ts, cnt)`) and then calls `c.next()` before invoking an operation that might delete `ptr`.

If the destructor of the class that is about to be removed has to delete a substructure, the safest thing to do is to record the addresses of all the objects to be deleted during the iteration and then delete them after the iteration is complete. Note that you should still call

```
db->record_new_object_location(ptr, 0xffffff);
```

during the iteration so that `os_database::fix_pointers()` will set to NULL any pointers that it finds to instances of `NotNeeded`.

After schema evolution is done, be sure to save the database.

Changing the Definition of a Base Class

If you change the definition of a class from which other classes are derived in a way that affects class layout, you must perform schema evolution for instances of each derived class as well as for instances of the base class. That is, changing the layout of a base class effectively changes the layout of all its derived classes, so they, too, must be evolved.

For example, suppose you have the following class hierarchy:

```
class Base {
private:
  int x;
  char* y
};

class Derived1 : public Base {
};

class Derived2 : public Base {
};
```

Then you decide to add a data member to `Base`:

```
class Base {
private:
  int x;
  char* y;
  float new_data_member;
};
```

You must do the following to evolve your PSE Pro databases:

- Add a schema evolution constructor to `Base`, `Derived1`, and `Derived2`. The latter two schema evolution constructors do not need to do anything other than invoke `Base`'s schema evolution constructor.
- Mark `old_Base`, `old_Derived1`, and `old_Derived2` (the latter two derived from `old_Base`) as persistent.
- Rename each class in the database.

Allocate new instances of each while iterating using the `os_object_cursor`.

Schema Evolution API Reference

The schema evolution API comprises the following functions:

os_database::rename_type()

`void database::rename_type(const char* old, const char* nw);`

This function renames all instances of type *`old`* to the new type *`nw`*.

os_database::record_new_object_location()

`void os_database::record_new_object_location (void* old, void* nw);`

This function is called as part of the schema evolution constructor. It maps instances of the *`old`* type to instances of the *`nw`* type. This allows all pointers in the database that point to the old instances to be evolved (that is, relocated) to point to the new instances.

The actual change of the pointers in the database happens during `os_ database::fix_pointers()`.

os_database::fix_pointers()

`void os_database::fix_pointers(os_unsigned_int32 n_pages=0);`

This function evolves, or relocates, all pointers in the database that have been recorded during the evolution of a database using the schema evolution constructor.

During this function, every nonempty page containing user data has to be visited to adjust the pointers of evolved objects. For large databases, this requires a lot of swap space. If *`n_pages`* is greater than `0`, `fix_pointers()` saves the database every time *`n_pages`* have been visited and calls `objectstore::return_all_pages()` to release swap space.

If *`n_pages`* is `0`, `fix_pointerS()` does *not* save the database.

Index