ObjectStore® PSE Pro™

Writing Applications with PSE Pro for C++
A Guide for the First-Time User

Release 6.3

PROGRESS SOFTWARE | Real Time Division

Writing Applications with PSE Pro for C++, Release 6.3, October 2005

© 2005 Progress Software Corporation. All rights reserved.

Progress® software products are copyrighted and all rights are reserved by Progress Software Corporation. This manual is also copyrighted and all rights are reserved. This manual may not, in whole or in part, be copied, photocopied, translated, or reduced to any electronic medium or machine-readable form without prior consent, in writing, from Progress Software Corporation.

The information in this manual is subject to change without notice, and Progress Software Corporation assumes no responsibility for any errors that may appear in this document.

The references in this manual to specific platforms supported are subject to change.

A (and design), Allegrix, Allegrix (and design), Apama, Business Empowerment, DataDirect (and design), DataDirect Connect, DataDirect Connect OLE DB, DirectAlert, EasyAsk, EdgeXtend, Empowerment Center, eXcelon, Fathom,, IntelliStream, O (and design), ObjectStore, OpenEdge, PeerDirect, P.I.P., POSSENET, Powered by Progress, Progress, Progress Dynamics, Progress Empowerment Center, Progress Empowerment Program, Progress Fast Track, Progress OpenEdge, Partners in Progress, Partners en Progress, Persistence, Persistence (and design), ProCare, Progress en Partners, Progress in Progress, Progress Profiles, Progress Results, Progress Software Developers Network, ProtoSpeed, ProVision, SequeLink, SmartBeans, SpeedScript, Stylus Studio, Technical Empowerment, WebSpeed, and Your Software, Our Technology-Experience the Connection are registered trademarks of Progress Software Corporation or one of its subsidiaries or affiliates in the U.S. and/or other countries. AccelEvent, A Data Center of Your Very Own, AppsAlive, AppServer, ASPen, ASP-in-a-Box, BusinessEdge, Cache-Forward, DataDirect, DataDirect Connect64, DataDirect Technologies, DataDirect XQuery, DataXtend, Future Proof, ObjectCache, ObjectStore Event Engine, ObjectStore Inspector, ObjectStore Performance Expert, POSSE, ProDataSet, Progress Business Empowerment, Progress DataXtend, Progress for Partners, Progress ObjectStore, PSE Pro, PS Select, SectorAlliance, SmartBrowser, SmartComponent, SmartDataBrowser, SmartDataObjects, SmartDataView, SmartDialog, SmartFolder, SmartFrame, SmartObjects, SmartPanel, SmartQuery, SmartViewer, SmartWindow, WebClient, and Who Makes Progress are trademarks or service marks of Progress Software Corporation or one of its subsidiaries or affiliates in the U.S. and other countries. Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries. Any other trademarks or trade names contained herein are the property of their respective owners.

September 2005

Contents

Preface 5

Writing PSE Pro for C++ Applications 9

The PSE Pro API 10

Database Management 11

Database Roots 11

Persistent Objects 11

Transaction Management 12

A Checklist of PSE Pro Operations 12

Flaws in the iostreams Version 19

PSE Pro Version 19

Building PSE Pro Applications 26

Generating Schemas 26

Compiling and Linking 27

Using the Collections Facility 28

An Example 29

Building the Collections Example 34

Compiler Options 36

get_os_typespec() 36

objectstore::initialize() 37

::operator delete() 37

os_collection::initialize() 38

os_database::close() 38

os_database::create_root() 39

os_database::of() 39

os_database::open() 39

os_database_root::get_value() 40

os_database_root::set_value() 40

os_Dictionary::pick() 40

OS_PSE_END_FAULT_HANDLER 41

OS_PSE_ESTABLISH_FAULT_HANDLER 41

OS_MARK_DICTIONARY() 41

OS_MARK_SCHEMA_TYPE() 41
os_transaction::begin() 42
os_transaction::commit() 42
os_typespec::get_char() 42
pssg 43

Index 45

Preface

Purpose *Writing Applications with PSE Pro for C++* shows how to write and build simple database applications that use basic features of the PSE Pro for C++ application programming interface (API), including the collections facility.

Audience This manual is written for C++ programmers with little or no experience with PSE Pro.

Release This manual supports PSE Pro for C++ Release 6.3.

How This Book Is Organized

The book is organized in sections, as follows:

- Making Complex Objects Persistent on page 10 discusses the benefits of using PSE Pro to make objects persistent.
- Basic PSE Pro Operations on page 11 describes the basic operations that any PSE Pro application must perform.
- A Checklist of PSE Pro Operations on page 12 lists the PSE Pro operations that you must implement in your application and the APIs that you use to implement them.
- Using PSE Pro: An Example Application on page 14 presents and compares two versions of an example program — one using `iostreams` and the other using PSE Pro.
- Building PSE Pro Applications on page 26 describes how to build an PSE Pro application. It uses the program presented in the previous section as an example.
- Using the Collections Facility on page 28 provides an overview of the collections facility and shows how to use some of its features. It uses a revised version of the PSE Pro program presented earlier to illustrate how to implement the collections facility.
- An Abbreviated PSE Pro Reference on page 36 provides a summary reference of all features of the API mentioned in this Guide. For detailed and complete information, refer to the*PSE Pro for C++ API User Guide* and to the *PSE Pro for C++ Collections Guide and Reference*.

Example Programs

The example programs discussed in this manual are available online in the directory `examples\restaurant` located in the PSE Pro installation directory, which by default on Windows platforms is `\odi\PSEPROC63`. On UNIX platforms, the default PSE Pro installation directory is `/opt/ODI/PSEPROC63`.

The `restaurant` directory consists of subdirectories that contain the source files for each program, as well as `README` files, makefiles, and scripts for building and running the programs. Each program is in its own subdirectory.

The example programs are not intended to be performance models but rather to illustrate how to use basic features of PSE Pro. For example, none of the programs executes more than one transaction at run time. But PSE Pro's *hot cache* architecture achieves its best performance as the application processes more and more transactions. Realistic applications would be designed to take advantage of this architecture.

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
```

Writing PSE Pro for C++ Applications

This Guide explains how to write simple PSE Pro applications in C++, focusing on the fundamental concepts and features of PSE Pro. You will learn the basics of

- Persistence
- Basic PSE Pro operations, such as transactions and database management
- Writing and building a PSE Pro application
- Using the collections facility

Note

One purpose of this Guide is to expose the details of how you write simple PSE Pro applications. The Guide therefore does not include information about automating development with the `make` utility. However, the on-line directories that contain the source files for the example programs also include makefiles; for more information, see Example Programs on page 5.

Making Complex Objects Persistent

PSE Pro for C++ is one of a broad and varied range of ObjectStore products that meet your database needs. The core competence of these products, however, is quite simple: making complex objects *persistent.* PSE Pro enables an application to store and retrieve objects in a database in the same form it uses to manipulate *transient* objects (that is, objects resident in program memory). When the application accesses objects stored in a PSE Pro database, they are ready for use and require no mapping code to move them from disk-resident format to memory-resident format.

The PSE Pro API

To write a PSE Pro application, you use the application programming interface (API) that is part of PSE Pro. The API consists of C++ classes and macros that perform certain PSE Pro operations, such as opening a database or defining a transaction. Using the API does not require learning another programming language, and your program source files do not need special preprocessing before compilation. You write all code in C++.

Many of the classes are designed for specialized tasks and are unnecessary for most applications. In fact, only a small subset of the functions and macros in the API are needed to perform the basic operations that are common to most PSE Pro applications.

This Guide describes the basic PSE Pro operations and shows how to use the API to implement them. All features of the API mentioned in this Guide are described in the last section, An Abbreviated PSE Pro Reference on page 36.

Basic PSE Pro Operations

To use PSE Pro, you must implement these basic operations in your application:

- Create or open the database in which data will be stored
- Create a database root
- Create and destroy objects in persistent memory
- Define transaction boundaries for accessing persistent data

Each of these operations is described in the following sections.

Database Management

The most commonly performed operations on a PSE Pro database are familiar:

- Creating the database
- Opening the database
- Closing the database

What might be less obvious is that a PSE Pro database is really a unit of persistent storage. This means, for example, that whenever you create a persistent object, you are creating it *in* a particular database. Therefore, all manipulations of a persistent object occur within the context of an *open* database. It is only when you are finished working with persistent objects that you can close the database.

Database Roots

In a PSE Pro application, you access the first object in a database by retrieving a *database root*. You create the root by using the PSE Pro API, giving it a name that you can later use to retrieve the root. You then associate the root with an object stored in your database — the *entry-point object*. Having the entry-point object gives you access to other objects, either through some form of associative access (such as by using a key value) or by navigational means.

Persistent Objects

When creating and destroying objects in persistent memory, you use the same operators — `new` and `delete` — as you would when creating objects in heap-allocated (that is, transient) memory. The difference is that when you create a persistent object, you are adding the object to the PSE Pro database, extending its lifetime beyond the lifetime of the application that created it. And when you destroy it, you are removing it from the database.

PSE Pro provides an overloading of `new` to create persistent objects and an overriding of `delete` to destroy them. PSE Pro can detect at run time whether you are using `new` to create a persistent or a heap-allocated object; likewise, it can detect at run time whether you are using `delete` to release persistent or heap-allocated storage. For information about these operators, see `::operator new()` on page 38 and `::operator delete()` on page 37.

Transaction Management

If operations on a database need to be atomic, PSE Pro applications can access persistent data from within a *transaction*. Briefly, a transaction is a sequence of statements in an application that read and write to persistent data. A transaction is atomic — that is, either all of its changes to the database are recorded or none of them. If an event prevents a transaction from completing, the transaction aborts, and all intermediate changes are rolled back to their pretransaction state. If the transaction successfully completes, all changes to the database are committed and made permanent.

Using transactions to access persistent data is optional in PSE Pro. The PSE Pro applications in this Guide use transactions.

A Checklist of PSE Pro Operations

This section lists the items you must implement when you write an application to use PSE Pro. These items include the major operations described in Basic PSE Pro Operations on page 11 as well as minor but equally essential items, such as including PSE Pro header files.

- Include the required PSE Pro header files. All PSE Pro applications must include `ostore.hh`. For more specialized features of the API, you might have to include other PSE Pro header files. The *PSE Pro for C++ API User Guide* lists the required header files for all classes that require them. For an example, see `restaurant.hh` on page 20, line 5.
- Include all code that performs any PSE Pro operation within the following macros:
  - `OS_PSE_ESTABLISH_FAULT_HANDLER`
  - `OS_PSE_END_FAULT_HANDLER`

  PSE Pro uses these macros to detect references to persistent memory.

  The simplest way to use these macros is to insert `OS_PSE_ESTABLISH_FAULT_HANDLER` at the beginning of `main()` or `WinMain()` and to insert `OS_PSE_END_FAULT_HANDLER` at the end. For more information about the macros, see `OS_PSE_ESTABLISH_FAULT_HANDLER` on page 41. For an example, see `init_db.cpp` on page 22, line 11 and line 33.
- Initialize PSE Pro by calling `objectstore::initialize()` before performing any PSE Pro operation. The simplest place to call this function is just after the `OS_PSE_ESTABLISH_FAULT_HANDLER` macro. For more information, see `objectstore::initialize()` on page 37. For an example, see `init_db.cpp` on page 22, line 12.
- Perform the necessary database operations, including creating, opening, and closing a database. We provide the following methods to perform these operations:

  - `os_database::create()` on page 39. For an example, see `init_db.cpp` on page 22, line 21.
  - `os_database::open()` on page 39. For an example, see `main.cpp` on page 24, line 21.
  - `os_database::close()` on page 38. For an example, see `main.cpp` on page 24, line 44.

- If your application uses transactions, define the transaction boundaries of each sequence of statements that operate on persistent data as a logical unit. PSE Pro defines these boundaries with calls to `os_transaction` methods, as described in `os_transaction::begin()` on page 42 and `os_transaction::commit()` on page 42. For an example, see `main.cpp` on page 24, line 24 and line 43.
- Create and destroy persistent data. To create a persistent object, use the PSE Pro overloading of operator `new`, as described in `::operator new()` on page 38. For examples, see `restaurant.cpp` on page 20, line 25 (an object) and line 52 (a character array).

  PSE Pro overrides operator `delete`, so the same syntax can be used to destroy persistent or transient objects; see `::operator delete()` on page 37.
- Create a database root and associate it with the entry-point object in your database. PSE Pro provides the following methods for managing database roots:
  - `os_database::create_root()` on page 39. For an example, see `restaurant.cpp` on page 20, line 27.
  - `os_database::find_root()` on page 40. For an example, see `restaurant.cpp` on page 20, line 39.
  - `os_database_root::get_value()` on page 40. For an example, see `restaurant.cpp` on page 20, line 41.
  - `os_database_root::set_value()` on page 40. For an example, see `restaurant.cpp` on page 20, line 29.

Using PSE Pro: An Example Application

To illustrate how to use PSE Pro in an application, this section examines a program that processes restaurant reservations. The program is implemented in two different versions:

- An `iostreams` version, which uses `iostreams` to store and retrieve data about a restaurant; see iostreams Version on page 15.
- A PSE Pro version, which uses PSE Pro to make persistent the object that contains the restaurant data; see PSE Pro Version on page 19.

Both versions are functionally identical: the user invokes the main application (`reserve`) and enters the number of persons for the dinner reservation; and the program either confirms the reservation by displaying the number of tables that have been reserved or, if there are not enough tables, denies the reservation. Likewise, both versions use an object of the class `Restaurant` to hold information about one restaurant. This information is updated each time `reserve` is invoked and successfully makes a reservation.

Each version of the program consists of these source files:

- `restaurant.hh`, the header file that declares the `Restaurant` class
- `restaurant.cpp`, the implementations file that defines the function members of the `Restaurant` class
- `init_db.cpp`, the source file for a utility application (`init_db`), which initializes the restaurant data
- `main.cpp`, the source file for the main application (`reserve`), which processes reservations

The similarities between the two versions will help to focus on the key differences in the PSE Pro version — that is, the basic PSE Pro operations. Commentary on the programs follows the listings of `init_db.cpp` and `main.cpp` for both versions.

iostreams Version

The `iostreams` version of the reservations program stores the restaurant data (the name of the restaurant and the number of available tables) in an ASCII file. The source files for the program are in the directory `examples\restaurant\iostreams` which is located in the PSE Pro installation directory. By default, on Windows platforms, this is `odi\PSEPROC63`; on UNIX platforms, `/opt/ODI/PSEPROC63`.

restaurant.hh

Here is the header file for the `Restaurant` class:

```
// restaurant.hh: defines the Restaurant class
#include <iostream>
#include <fstream>
#include <string.h>
#include <stdlib.h>

class Restaurant {
private:
  const char* name; // name of restaurant
  int tables; //number of tables available (4 persons per table)
  static const int BUFSIZE;
  static const int PERSONSPERTABLE;
  char* dupl_string(const char* s);
public:
  Restaurant(const char* s, int t);
  ~Restaurant() { delete [] (char*)name; };
  static void create_restaurant(const char* s,int t,ofstream* db);
  static Restaurant* get_restaurant(fstream* db);
  void save_restaurant(fstream* db);
  const char* get_name() { return name; }
  int get_tables() { return tables; }
  void set_tables(int t) { tables = t; }
  int make_reservation(int n_persons);
private:
  // declared but unimplemented functions to prevent
  // inadvertent calls
  Restaurant();
  Restaurant(const Restaurant&);
  Restaurant& operator=(const Restaurant&);
};
```

restaurant.cpp

Here is the implementation file:

```
// restaurant.cpp: implements members of Restaurant
// and defines globals
#include "restaurant.hh"

#include "dbname.h";

const int Restaurant::BUFSIZE = 100;
const int Restaurant::PERSONSPERTABLE = 4;

// create a Restaurant object
Restaurant::Restaurant(const char* s, int t):
  name(dupl_string(s))
{
  tables = t;
}
```

```
// save name of restaurant and the current number of tables
void Restaurant::save_restaurant(fstream* db)
{
  db->seekp(0, ios::beg);
  *db << dec << tables << endl;
  *db << name << endl;
}

// store name of restaurant (s) and initial number of tables (t)
// in the file that db points to
void Restaurant::create_restaurant(const char* s, int t,
  ofstream* db)
{
  *db << dec << t << endl;
  *db << s << endl;
}

// return a pointer to a Restaurant object that is
// initialized from the contents of the database that
// db points to
Restaurant* Restaurant::get_restaurant(fstream* db)
{
  char name_buf[BUFSIZE];
  char table_buf[BUFSIZE];

  db->getline(table_buf, BUFSIZE);
  db->getline(name_buf, BUFSIZE);

  return(Restaurant*)new Restaurant(name_buf,atoi(table_buf));
}

// allocate storage to hold the string in s, copy s to the
// newly allocated storage, and return a pointer to the storage
char* Restaurant::dupl_string(const char* s)
{
  int len = strlen(s)+1;
  char* p = new char [len];
  return strcpy(p, s);
}

// return the number of tables reserved, based on the number of
// persons in the party (n_persons); otherwise, 0
int Restaurant::make_reservation(int n_persons)
{
  int tables_to_reserve = 0;
  int tables_available = get_tables();

  if (!tables_available)
    ;  // do nothing, all booked up!
  else if (n_persons <= PERSONSPERTABLE) {
    set_tables(tables_available-1);
    tables_to_reserve = 1;
  }
  else { // figure out how many tables to reserve
    tables_to_reserve = n_persons / PERSONSPERTABLE;
    if (tables_to_reserve*PERSONSPERTABLE < n_persons)
      tables_to_reserve++;
    if (tables_to_reserve > tables_available)
      // not enough tables, so can't make reservation
      tables_to_reserve = 0;
```

```
    else
      // successful reservation
      set_tables(tables_available-tables_to_reserve);
  }
  return tables_to_reserve;
}
```

init_db.cpp

Here is the source file for `init_db`, the application that creates a streams file and initializes it with the restaurant name and number of tables specified on the command line. Commentary follows the listing.

```
// init_db.cpp: creates a restaurant object and writes its data
// members (name and number of tables) to a file
#include "restaurant.hh"

#include "dbname.h";

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

  // check for missing or bad arguments
  if (argc != 3 || !(n_tables = atoi(argv[2]))) {
    cerr << "USAGE: init_db <restaurant-name> <n-tables>\n";
    return 1;
  }

  // create file for storing restaurant data
  ofstream db(DB_NAME);

  // create a restaurant
  Restaurant::create_restaurant(argv[1], n_tables, &db);

  db.close();

  return 0;
}
```

Commentary on init_db.cpp

The `init_db` application does the following:

1. Creates a streams file named `restaurants.db`.
2. Calls create_restaurant() to store information about the restaurant in the file. The information to be stored is supplied on the command line. The `create_restaurant()` function does not create a `Restaurant` object, which would only be destroyed once `init_db` finishes executing.
3. Closes the file.

Command lines for compiling, linking, and running

Here are the command lines for compiling and linking `init_db` on Windows platforms:

```
cl /c init_db.cpp restaurant.cpp
cl /Feinit_db.exe init_db.obj restaurant.obj
```

The following command line creates the file `restaurants.db` and stores the two arguments, `Il Falchetto` and `10`:

```
init_db "Il Falchetto" 10
```

main.cpp

Here is the source file for the main application (`reserve`), which processes restaurant reservations. Commentary follows the listing.

```
// main.cpp: processes restaurant reservations
#include "restaurant.hh"

#include "dbname.h";

int main(int argc, char** argv)
{
  int tables_reserved, n_persons;

  // check for missing or bad arguments
  if (argc != 2 || !(n_persons = atoi(argv[1]))) {
    cerr << "USAGE: reserve <n-persons>\n";
    return 1;
  }

  // open file for reading restaurant data and writing
  // updated data
  fstream db(DB_NAME, ios::in|ios::out);

  // get a Restaurant object, initialize it from values in file
  Restaurant* restaurant = Restaurant::get_restaurant(&db);

  // book a reservation
  tables_reserved = restaurant->make_reservation(n_persons);

  // confirm or deny reservation
  if (tables_reserved)
    cout << "Reserved " << tables_reserved << " tables";
  else
    cout << "Sorry, all booked";
  cout << " at " << restaurant->get_name() << endl;

  // save restaurant data, then delete restaurant object
  restaurant->save_restaurant(&db);
  delete restaurant;

  db.close();

  return 0;
}
```

Commentary on main.cpp

The main application (`reserve`) does the following:

1. Opens `restaurants.db` for reading and writing.
2. Calls `get_restaurant()` to return a pointer to the object that has been constructed from the information stored in `restaurants.db`. This step invokes the constructor, `Restaurant()`, which also assigns its file pointer argument (`db`) to the `database` member. The destructor needs this member to update the file with the information supplied from the `tables` and `name` data members.
3. Calls `make_reservation()` with the number of persons for which the reservation is being made. This method also updates the `tables` member of the `Restaurant` object.

4. Confirms or denies the reservation based on the availability of a sufficient number of tables.
5. Calls `save_restaurant()`, which writes the updated number of tables as well as the restaurant name back out to file. The call to `fstream::seekp()` inside `save_restaurant()` ensures that the updates overwrite the file rather than append to its end.
6. Closes `restaurants.db`.

Command lines for compiling, linking, and running

Here are the command lines to compile and link the main application:

```
cl /c main.cpp restaurant.cpp
cl /Fereserve.exe main.obj restaurant.obj
```

The following is a sample run:

```
C:\> reserve 11
Reserved 3 tables at Il Falchetto
```

Flaws in the iostreams Version

There are several serious flaws in the `iostreams` version of the restaurant reservations program. These flaws become more apparent as we revise the program to make it more realistic — for example, by adding more data members to the `Restaurant` class or by modifying the program to manage reservations for more than one restaurant:

- The program is not fail-safe. An exception or any other abnormal condition that causes the program to crash will prevent `save_restaurant()` from updating `restaurants.db`.
- As more classes are added to the program — classes with data members might have to be stored and retrieved — writing code to serialize and deserialize the objects becomes more complicated and cumbersome. This limitation also becomes more apparent as more data members are added to the `Restaurant` class and as the relationships between objects become more complex.
- The program does not scale. If the program were required to process thousands of `Restaurant` objects, relying on `iostreams` to store, retrieve, and update `Restaurant` data members would slow performance to the point of making the program unusable.

In the next section, we take the first steps toward making the program more usable by implementing it to use PSE Pro.

PSE Pro Version

The PSE Pro version of the reservations program makes an object of the class `Restaurant` persistent. The `reserve` application does not construct a `Restaurant` object each time it is invoked. Rather, it opens the database and accesses the object that is already there, ready for use. Any updates that the application makes to the `tables` data member are persistent because they are updates to the object, which is itself persistent.

The source files for the program are in the directory `examples\restaurant\basic_os`which is located in the PSE Pro installation directory. By default, this is `odi\PSEPROC63`.

restaurant.hh — Here is the header file about the program for the `Restaurant` class. (Line numbers are included in this file and in the remaining files for the convenience of cross-referencing.)

```
01  // restaurant.hh: defines the Restaurant class
02  #include <iostream>
03  #include <string.h>
04  #include <stdlib.h>
05  #include <os_pse/ostore.hh>
06
07  class Restaurant {
08  private:
09    const char* name; // name of restaurant
10    int tables; //number of tables available (4 persons per table)
11    static const int PERSONSPERTABLE;
12    static const char* DB_ROOT; // name of database root
13    char* dupl_string(const char* s);
14  public:
15    Restaurant(const char* s, int t);
16    ~Restaurant() { delete [] (char*)name; }
17    static void create_restaurant(const char* s, int t,
18      os_database* db);
19    static Restaurant* get_restaurant(os_database* db);
20    const char* get_name() { return name; }
21    int get_tables() { return tables; }
22    void set_tables(int t) { tables = t; }
23    int make_reservation(int n_persons);
24
25    // retrieve a pointer to the Restaurant typespec object
26    static os_typespec* get_os_typespec(){ return os_ts<Restaurant>::get(); }
27
28  private:
29    // declared but unimplemented functions to prevent
30    // inadvertent calls
31    Restaurant();
32    Restaurant(const Restaurant&);
33    Restaurant& operator=(const Restaurant&);
34  };
```

restaurant.cpp — Here is the implementation file.

```
01  // restaurant.cpp: implements members of Restaurant
02  // and defines globals
03  #include "restaurant.hh"
04
05  #include "dbname.h";
06
07  const int Restaurant::PERSONSPERTABLE = 4;
08  const char* Restaurant::DB_ROOT = "restaurant_root";
09
10  // create a Restaurant object
11  Restaurant::Restaurant(const char* s, int t):
12    name(dupl_string(s))
```

```
13  {
14    tables = t;
15  }
16
17  // create a Restaurant object and a database root, and associate
18  // the two
19  void Restaurant::create_restaurant(const char* s, int t,
20    os_database* db)
21  {
22    // get a pointer to typespec for Restaurant class
23    os_typespec* ts = Restaurant::get_os_typespec();
24    // create Restaurant object
25    Restaurant* restaurant = new(db, ts) Restaurant(s, t);
26    // create database root
27    os_database_root* db_root = db->create_root(DB_ROOT);
28    // associate root with object
29    db_root->set_value(restaurant, ts);
30
31
32  }
33
34  // return a pointer to the Restaurant object that is associated with
35  // the database root
36  Restaurant* Restaurant::get_restaurant(os_database* db)
37  {
38    // find root
39    os_database_root* db_root = db->find_root(DB_ROOT);
40    // retrieve and return associated object
41    return (Restaurant*)db_root->get_value(
42      Restaurant::get_os_typespec());
43
44
45  }
46
47  // allocate storage to hold the string in s, copy s to the
48  // newly allocated storage, and return a pointer to the storage
49  char* Restaurant::dupl_string(const char* s)
50  {
51    int len = strlen(s)+1;
52    char* p = new (os_database::of(this),
53      os_typespec::get_char(), len) char [len];
54    return strcpy(p, s);
55  }
56
57  // return the number of tables reserved, based on the number of
58  // persons in the party; otherwise, 0
59  int Restaurant::make_reservation(int n_persons)
60  {
61    int tables_to_reserve = 0;
62    int tables_available = get_tables();
63
64    if (!tables_available)
65      ;  // do nothing, all booked up!
66    else if (n_persons <= PERSONSPERTABLE) {
67      set_tables(tables_available-1);
68      tables_to_reserve = 1;
69    }
```

```
70    else { // figure out how many tables to reserve
71      tables_to_reserve = n_persons / PERSONSPERTABLE;
72      if (tables_to_reserve*PERSONSPERTABLE < n_persons)
73        tables_to_reserve++;
74      if (tables_to_reserve > tables_available)
75        // not enough tables, so can't make reservation
76        tables_to_reserve = 0;
77      else
78        // successful reservation
79        set_tables(tables_available-tables_to_reserve);
80    }
81    return tables_to_reserve;
82  }
```

init_db.cpp

Here is the source file for `init_db`, the application that creates a `Restaurant` object and makes it persistent. Commentary follows the listing.

```
01  // init_db.cpp: creates database for restaurant reservations
02  // program
03  #include "restaurant.hh"
04
05  #include "dbname.h";
06
07  int main(int argc, char** argv)
08  {
09    int n_tables;
10
11    OS_PSE_ESTABLISH_FAULT_HANDLER {
12      objectstore::initialize();
13      os_transaction::initialize();
14      // check for missing or bad arguments
15      if (argc != 3 || !(n_tables = atoi(argv[2]))) {
16        cerr << "USAGE: init_db <name> <n-tables>\n";
17        return 1;
18      }
19
20      // create database for storing a Restaurant object
21      os_database* db = os_database::create(DB_NAME);
22
23      // begin update transaction
24      os_transaction* txn =
25        os_transaction::begin("the_txn", os_transaction::update);
26        // create a Restaurant object
27        Restaurant::create_restaurant(argv[1], n_tables, db);
28
29      os_transaction::commit();
30      db->close();
31      db = NULL;
32
33    } OS_PSE_END_FAULT_HANDLER
34
35    return 0;
36  }
```

Commentary on init_db.cpp

The structure and interface of the PSE Pro version of `init_db` is nearly the same as that of the `iostreams` version, except that it stores the `Restaurant` *object* — not the values of its data members — in a database. This version does the following:

1 Creates a database for the `Restaurant` object.

2 Calls `create_restaurant()` to store restaurant data.

3 Closes the database.

The differences, however, are significant in that they apply to any application that uses PSE Pro. The key differences are

- All code that performs PSE Pro operations is delimited by page-fault handling macros.
- PSE Pro is initialized by a call to `objectstore::initialize()`.
- In this application, all code that accesses persistent data is contained within a PSE Pro transaction. The PSE Pro transaction facility is initialized by a call to `os_transaction::initialize()`. The transaction starts with a call to `os_transaction::begin()` and ends with a call to `os_transaction::commit()`.

The underlying implementation is also different, as a comparison of the two versions of `restaurant.cpp` will show. Recall that the `iostreams` version of the `create_restaurant()` method does not create an object but simply writes two data values to a file. The PSE Pro version, on the other hand, does create a `Restaurant` object and makes it persistent, storing it in a database. That is, the object — not just the values of its data members — is now accessible to other applications when they open the database.

The `create_restaurant()` method uses the PSE Pro overloading of `::operator new()` to create the `Restaurant` object. Persistent `new` has two arguments that transient `new` does not have:

- The placement argument (`db`), which points to the persistent database that was opened in `init_db.cpp`.
- The typespec argument (`ts`), which points to an `os_typespec` object. This argument supplies PSE Pro with type information it needs when allocating persistent storage. The pointer is obtained by calling `get_os_typespec()`, which is declared as a method of `Restaurant` and uses PSE Pro's `os_ts<>::get()` method.

For a description of persistent `new`, see `::operator new()` on page 38.

The PSE Pro version of `create_restaurant()` also creates a database root and then associates the root with the `Restaurant` object created earlier. Here are the statements that perform both operations:

```
os_database_root* db_root = db->create_root(DB_ROOT);
db_root->set_value(restaurant, ts);
```

The `ts` argument is the same typespec that was specified in `::operator new()` when the `Restaurant` object was created. Although this argument is not required in calls to `set_value()`, ObjectStore Technical Support recommends using it to prevent mismatch errors; see `os_database_root::set_value()` on page 40.

Applications can access the object by retrieving the root with which it is associated, using the name (`DB_ROOT`) with which it was created.

Note — `init_db` application does not — and *must not* — delete the `Restaurant` object created by `create_restaurant()`. The program stores this object persistently; if the object were deleted, it would no longer exist in the database.

main.cpp — Here is the source file for the main application (`reserve`), which processes restaurant reservations. Commentary follows the listing.

```
01  // main.cpp: processes restaurant reservations
02  #include "restaurant.hh"
03  #include <stdlib.h>
04
05  #include "dbname.h";
06
07  int main(int argc, char** argv)
08  {
09    int tables_reserved, n_persons;
10
11    OS_PSE_ESTABLISH_FAULT_HANDLER {
12      objectstore::initialize();
13      os_transaction::initialize();
14      // check for missing or bad argument
15      if (argc != 2 || !(n_persons = atoi(argv[1]))) {
16        cerr << "USAGE: reserve <n-persons>\n";
17        return 1;
18      }
19
20      // open database
21      os_database* db = os_database::open(DB_NAME);
22
23      // begin update transaction
24      os_transaction* txn =
25        os_transaction::begin("the_txn", os_transaction::update);
26        // get Restaurant object from database
27        Restaurant* restaurant =
28          Restaurant::get_restaurant(db);
29
30        // book a reservation
31        tables_reserved =
32          restaurant->make_reservation(n_persons);
33
34        // confirm or deny reservation
35        if (tables_reserved)
36          cout << "Reserved " << tables_reserved << " tables";
37        else
38          cout << "Sorry, all booked";
39        cout << " at " << restaurant->get_name() << endl;
40
41        // don't delete object! -- it's persistent
42
43      os_transaction::commit();
44      db->close();
45      db = NULL;
46
47    } OS_PSE_END_FAULT_HANDLER
48
49    return 0;
50  }
```

Commentary on main.cpp

The PSE Pro version of the main application is similar to the `iostreams` version in both structure and interface — except for the differences already noted in "Commentary on init_db.cpp" on page 22. In other words, the PSE Pro version follows the same fundamental steps for processing a reservation, but without explicitly saving the object and without deleting the persistent `Restaurant` object. When the transaction commits, all changes to the object are made permanent in the database.

Here are the steps implemented in `main.cpp`:

1. Open the database.
2. Call `get_restaurant()`.
3. Call `make_reservation()`.
4. Confirm or deny the reservation.
5. Close the database.

The implementation of `get_restaurant()` in Step 2 is very different from the `iostreams` version. Instead of using `new` to create a `Restaurant` object each time the application is invoked, the PSE Pro version retrieves the same `Restaurant` object that was previously created in persistent memory by `init_db`.

To retrieve the object, `get_restaurant()` first gets a pointer to the database root, using the name (the value of the string to which `DB_ROOT` points) with which it was created. Once it has the root, it calls a method on the root object to retrieve the associated entry-point object. The statements that do both operations are

```
os_database_root* db_root = db->find_root(DB_ROOT);
```

```
return (Restaurant*)db_root->get_value(
    Restaurant::get_os_typespec());
```

As in the call to `set_value()` that was used initially to associate the root with an object (see `restaurant.cpp` on page 20, line 29), the call to `get_value()` includes a typespec argument, only in this case the typespec is obtained by calling `get_os_typespec()` in the argument list. The typespec argument is optional but is recommended as a way to prevent mismatch errors. If you want PSE Pro to check the typespec argument, you must also specify it when you call `set_value()` to associate the root with the object, as shown in `restaurant.cpp` on page 20, line 29.

Preventing memory leaks

The `os_database` object to which `db` points is deleted when `os_database::close()` is invoked for `db`; see `main.cpp`on page 24, line 44. This is a transient object that *represents* the persistent database. Deleting it does *not* delete the database. Trying to use the `db` object after the database is closed throws the exception `os_err_database_closed`. For information about using the PSE Pro API for help in managing the lifetimes of transient objects, see Closing Databases in Chapter 3 of the *PSE Pro for C++ API User Guide.*

Building PSE Pro Applications

The steps for building a PSE Pro application are

1 Generate schemas.

2 Compile the source code.

3 Link the application.

Except for Step 1, building a PSE Pro application is similar to building any other application. The first step (*schema generation*) produces information about the types of the objects that will be made persistent. PSE Pro uses this schema information whenever an application accesses a database. Once you have generated the schema, you use it in the remaining steps of the build process, as described in Compiling and Linking on page 27.

Note

Before building an application, you must have installed PSE Pro for C++ and correctly set all necessary environment variables. In particular, the variables `INCLUDE` and `LIB` should contain the correct PSE Pro directories. By default these are `C:\ODI\PSEPROC\include` and `C:\ODI\PSEPROC\lib`, respectively.

Generating Schemas

Here are the steps for generating schemas:

1 Write a schema source file.

2 Compile the source file to check for syntax errors.

3 Run the PSE Pro schema generator, `pssg`, against the schema source file.

The following paragraphs describe each step in detail.

Writing the schema source file

The purpose of the schema source file is to mark the classes of objects that will be made persistent. Classes you must mark include not only those of objects stored directly in persistent memory (like the `Restaurant` class in the reservations program) but also any *reachable* classes. A class is reachable if it is the base class, or the class of a member, of a persistent object. For detailed information about the classes you must mark, see in Chapter 2 of *Building Applications with PSE Pro for C++*.

After identifying the classes for which you must generate a schema, you use the `OS_MARK_SCHEMA_TYPE()` macro to mark each class in the schema source file; for more information about this macro, see `OS_MARK_SCHEMA_TYPE()` on page 41.

In the restaurant reservations program, only one type of object, `Restaurant`, is persistently stored. Consequently, `Restaurant` is the only class that must be marked in the schema source file, as follows:

```
OS_MARK_SCHEMA_TYPE(Restaurant);
```

The schema source file must also include the PSE Pro header file `ostore.hh`, as well as the application header files that define the marked classes. Any other PSE Pro header files included in your application must also be included in the schema source file. See Using the Collections Facility on page 28 for an example that uses additional

PSE Pro header files. PSE Pro header files are order dependent, and `ostore.hh` must be included before any others.

Here is the schema source file (`schema.scm`) for the reservations program. Note that `ostore.hh` is already included in `restaurant.hh` and is, therefore, not included again.

schema.scm

```
#include "restaurant.hh"
OS_MARK_SCHEMA_TYPE(Restaurant);
```

Checking for syntax errors

The schema source file, like all other source files that you write, must contain valid C++ code. It is therefore recommended (but not required) that you compile it to check for syntax errors. Here is the command line to run the compiler to check the example schema source file for syntax errors:

```
cl /MD /c /Tpschema.scm
```

For information about the compiler options used in this command line, see Compiler Options on page 36. After successful compilation, you can delete the generated object file. It is not needed for building a PSE Pro application. The object file to be used for the application will be generated after you run the `pssg` schema-generation tool.

pssg command line

After you have a syntactically correct schema source file, you are ready to generate the schema object file. To do so, you use the `pssg` schema-generation tool, specifying the schema source file as one of its arguments. Here is the command line to generate schema for the reservations program:

```
pssg -asof schema.obj schema.scm
```

The `-asof` option lets you specify the name of the object file, in this case, `schema.obj`. If you do not use the `-asof` option, the default name of the object file is `osschm.obj`. Note that the schema source file (`schema.scm` in the example) does not require an option; however, it must appear at the end of the command line. For information about other `pssg` options, see `pssg` on page 43.

After processing, `pssg` produces the application schema object file, `schema.obj`. This object file is ready for linking into your application; see Compiling and Linking on page 27.

Note

Because `pssg` produces a compiled object file (in the example, `schema.obj`), the command line must include the same options you specify when compiling other PSE Pro source files; see Compiler Options on page 36.

Compiling and Linking

After you have successfully run `pssg`, you are ready to compile and link your application. Compiling and linking a PSE Pro application are straightforward tasks and can be made even more so by combining both operations on the same command line. They are separated here to clarify what is involved in each operation.

Compile line

Here is the command line for compiling the application source files:

```
cl /GX /MD /c main.cpp init_db.cpp restaurant.cpp
```

For information about the compiler options in this command line, see Compiler Options on page 36.

Link line

Here are the command lines for linking the application object files into the example executables:

```
cl /Feinit_db.exe init_db.obj restaurant.obj schema.obj
cl /Fereserve.exe main.obj restaurant.obj schema.obj
```

Note that the link lines must also include the application schema object file produced by `pssg`.

Except for the `/Fe` option, which specifies the name of the executable, no other linker options are required. `cl` automatically links the correct PSE Pro library, `osclt.lib` or `oscltd.lib` (debug version), with the application.

Running the application

The following is a sample run:

```
C:\> reserve 11
Reserved 3 tables at Il Falchetto
```

Using the Collections Facility

The PSE Pro version of the restaurant reservations program is still too limited to be a useful application. One major limitation is that it can store only one `Restaurant` object. One way to overcome this limitation is to redesign the program to use a C++ data structure such as a linked list or a tree, which would provide navigational access to additional objects in the database. See, for example, the sample program in the *PSE Pro for C++ Tutorial* which uses a binary tree to navigate among objects in the database.

But PSE Pro provides another way to store additional objects in the database — the collections facility. This facility enables applications to group database objects in a collection and then to perform queries on the collection.

Collection types

A *collection* is an object (such as a set or list) that can group other objects (such as `Restaurant` objects). PSE Pro provides five classes for creating a collection: `os_array`, `os_bag`, `os_Dictionary`, `os_list`, and `os_set`. (There are also templated versions of these classes, which provide better type safety.) Each class has a set of behaviors and characteristics, making it suitable for a range of applications. For information about the different collection types and how to determine which to use, see Choosing a Collection Type in Chapter 1 of the *PSE Pro for C++ Collections Guide and Reference*.

Populating a collection

After choosing a collection type, you typically create a persistent collection object, associate this object with a database root, and then populate the collection by inserting other objects in it. The inserted objects are the *elements* of the collection.

The elements in a populated collection are not actually the objects, but pointers to the objects. This means that an object can be an element of more than one collection at a time and that you can provide your own data structure for organizing and accessing objects that are also elements of a collection.

Once a collection is populated, you can perform different access and retrieval operations on it. For example, you can use a cursor to navigate through the collection or use a dictionary to quickly find elements in the collection using the dictionary's key.

Requirements

To use the collections API, an application must

- Include the `coll.hh` header file just after `ostore.hh`.
- For applications that use the `os_Dictionary` class, include `coll\dict_pt.hh` after `coll.hh` and, for source files that instantiate an `os_Dictionary`, include `coll\dict_pt.cc`.
- Call `os_collection::initialize()` just after `objectstore::initialize()`.
- For applications that use the `os_Dictionary` class specify information about it in the schema source file; see Building the Collections Example on page 34.

For other requirements that might apply, see Requirements for Collections Applications in Chapter 1 of the *PSE Pro for C++ Collections Guide and Reference*.

An Example

This section presents and discusses a revision of the restaurant reservations program. The revised program uses features of the collections facility to store and retrieve multiple persistent objects. It creates an `os_Dictionary` object to enable quickly looking up restaurants by name. To make the program more interesting, another data member (`city`) has been added to the `Restaurant` class, to identify the restaurant's location.

These are the source files for the main application, `reserve`:

- `restaurant.hh`, the header file
- `restaurant.cpp`, the implementations file
- `main.cpp`, the main application (`reserve`)

The source files for the program are in the directory `examples\restaurant\coll_os` which is located in the PSE Pro installation directory. By default, this is `odi\PSEPROC63`.

The source files also include the following additional programs:

- `init_db`, populates the database with several `Restaurant` objects
- `rlist`, lists all restaurants in the database or, if you supply a city argument, all the restaurants in the given city.
- `add_restaurant`, adds a Restaurant object to the database from arguments supplied on the command line

restaurant.hh

Here is the source listing for the header file, `restaurant.hh`:

```
01  // restaurant.hh: defines the Restaurant class
02  #include <iostream>
03  #include <string.h>
04  #include <stdlib.h>
05  #include <os_pse/ostore.hh>
```

```
06  #include <os_pse/coll.hh>
07
08  // forward declarations
09  class Restaurant;
10
11  // global function
12  ostream& operator<<(ostream& os, Restaurant& r);
13
14  class Restaurant {
15  private:
16    const char* name; // name of restaurant
17    const char* city; // name of city where it's located
18    int tables; // number of tables available (4 persons per table)
19    static const int PERSONSPERTABLE;
20    static const int QS_SIZE; // size of buffer holding query string
21    char* dupl_string(const char* s);
22  public:
23    Restaurant(const char* n, const char* c, int t);
24    ~Restaurant() { delete [] (char*)name; delete [] (char*)city; }
25    const char* get_name() { return name; }
26    const char* get_city() { return city; }
27    int get_tables() { return tables; }
28    void set_tables(int t) { tables = t; }
29    int make_reservation(int n_persons);
30    static Restaurant* create_restaurant(const char* n,
31      const char* c, int t, os_database* db);
32    static os_Dictionary<char*, Restaurant*>*
33       create_restaurant_dict(os_database* db);
34
35
36
37
38    // retrieve a pointer to the Restaurant typespec object
39    static os_typespec* get_os_typespec(){ return os_ts<Restaurant>::get(); }
40
41  private:
42     // declared but unimplemented functions to prevent
43    // inadvertent calls
44    Restaurant();
45    Restaurant(const Restaurant&);
46    Restaurant& operator=(const Restaurant&);
47  };
```

restaurant.cpp Here is the implementation file:

```
01  // restaurant.cpp: implements members of Restaurant
02  // and defines globals
03#include <os_pse/ostore.hh>
04#include <os_pse/coll.hh>
05#include <os_pse/coll/dict_pt.cc>
06
07  #include "restaurant.hh"
08
09  #include "dbname.h";
10  const char* DB_ROOT = "restaurant_root";
11
12  const int Restaurant::QS_SIZE = 100;
13  const int Restaurant::PERSONSPERTABLE = 4;
```

```
14
15
16  // create a Restaurant object
17  Restaurant::Restaurant(const char* n, const char* c, int t):
18    name(dupl_string(n)), city(dupl_string(c))
19  {
20    tables = t;
21  }
22
23  // return a pointer to a new Restaurant object
24  Restaurant* Restaurant::create_restaurant(const char* n,
25    const char* c, int t, os_database* db)
26  {
27    // get a pointer to typespec for Restaurant class
28    os_typespec* ts = Restaurant::get_os_typespec();
29    // create Restaurant object
30    return new(db, ts) Restaurant(n, c, t);
31  }
32
33    // Create a dictionary object and a root, and associate the two;
34  // return a pointer to the collection object
35  os_Dictionary<char*, Restaurant*>*
36      Restaurant::create_restaurant_dict(os_database* db)
37  {
38      os_Dictionary<char*, Restaurant*>* rest_dict;
39  // get a typespec for os_set
40    os_typespec* rest_ts =
41    os_Dictionary<char*, Restaurant*>::get_os_typespec();
42    // create a dictionary for Restaurant objects
43    rest_dict = new(db, rest_ts) os_Dictionary<char*, Restaurant*>;
44
45    // create a database root and associate it with the dictionary
46    os_database_root* db_root = db->create_root(DB_ROOT);
47    db_root->set_value(rest_dict, rest_ts);
48
49    return rest_dict;
50  }
51
52  // allocate storage to hold the string in s, copy s to the
53  // newly allocated storage, and return a pointer to the storage
54  char* Restaurant::dupl_string(const char* s)
55  {
56    int len = strlen(s)+1;
57    char* p = new(os_database::of(this), os_typespec::get_char(),
58    len) char [len];
59    return strcpy(p, s);
60  }
61
62  // return the number of tables reserved, based on the number of
63  // persons in the party; otherwise, 0
64  int Restaurant::make_reservation(int n_persons)
65  {
66    int tables_to_reserve = 0;
67    int tables_available = get_tables();
68
69    if (!tables_available)
70      ;  // do nothing, all booked up!
```

```
71  else if (n_persons <= PERSONSPERTABLE) {
72      set_tables(tables_available-1);
73    tables_to_reserve = 1;
74    }
75    else { // figure out how many tables to reserve
76      tables_to_reserve = n_persons / PERSONSPERTABLE;
77      if (tables_to_reserve*PERSONSPERTABLE < n_persons)
78        tables_to_reserve++;
79      if (tables_to_reserve > tables_available)
80        // not enough tables, so can't make reservation
81        tables_to_reserve = 0;
82    else
83        // successful reservation
84        set_tables(tables_available-tables_to_reserve);
85    }
86    return tables_to_reserve;
87
88  // overloading of << for displaying information about a restaurant
89  ostream& operator<<(ostream& os, Restaurant& r)
90  {
91    cout << r.get_name() << " in " << r.get_city()
92      << " has " << r.get_tables() << " table(s) available.\n";
93    return os;
94  }
```

main.cpp

Here is the source listing for main application, `reserve`. Commentary follows the listing.

```
01  // main.cpp: processes restaurant reservations
02  #include "restaurant.hh"
03  #include <stdlib.h>
04
05  #include "dbname.h";
06  extern const char* DB_ROOT;
07
08  int main(int argc, char** argv)
09  {
10    OS_PSE_ESTABLISH_FAULT_HANDLER {
11      int tables_reserved;
12
13      // check for missing or bad arguments
14      if (argc != 3) {
15        cerr << "USAGE: reserve <name> <n-persons>\n";
16        return 1;
17      }
18
19      // initialize ObjectStore and the collections and transaction facilities
20      objectstore::initialize();
21      os_collection::initialize();
22      os_transaction::initialize();
23      // open database
24      os_database* db = os_database::open(DB_NAME);
25
26      os_database_root* db_root;
27
28      // start update transaction
29      os_transaction* txn =
```

```
30        os_transaction::begin("the_txn", os_transaction::update);
31        // find the database root
32        db_root = db->find_root(DB_ROOT);
33
34        // retrieve dictionary of restaurants associated with root
35        os_Dictionary<char*, Restaurant*>* rdict =
36          (os_Dictionary<char*, Restaurant*>*)db_root->get_value(
37            os_Dictionary<char*, Restaurant*>::get_os_typespec());
38
39        Restaurant* r = rdict->pick(argv[1]);
40
41        // restaurant in database?
42        if (r) { // yes
43          // try to book a reservation
44          tables_reserved = r->make_reservation(atoi(argv[2]));
45
46          // confirm or deny reservation
47          if (tables_reserved)
48            cout << "Reserved " << tables_reserved
49              << " tables.\n";
50          else // not enough tables
51            cout << "Sorry, all booked.\n";
52        }
53        else // name not found in database
54          cout << "Can\'t find " << argv[1] << " in database.\n";
55
56      os_transaction::commit();
57      db->close();
58
59      db = NULL;
60      db_root = NULL;
61
62    } OS_PSE_END_FAULT_HANDLER
63
64    return 0;
65  }
```

Commentary on main.cpp

The commentary focuses on the following aspects of the program:

- The use of a collection as the entry-point object
- The use of a dictionary key to retrieve `Restaurant` objects

Collection as entry-point object

The non-collections version of the program stored one `Restaurant` object in its database and retrieved that object (the entry-point object) by associating it with the root; see the noncollections version of `restaurant.cpp` on page 20, lines 25-29. In the collections version, the entry-point object is a collection, which can be populated with multiple `Restaurant` objects. (The task of populating the collection is handled by `init_db.cpp`, which is included with the on-line source files for the collections version of the program but is not listed here.)

The `Restaurant::create_restaurant_dict()` function (see the collections version of `restaurant.cpp` on page 30, line 35) does the work of creating the entry-point object and the root, and associating the two, as follows:

- It creates a persistent collection object (`rest_dict`) using persistent `new` on line 43. The key used by `rest_dict` contains the names of the restaurants.

- It gets a pointer to a newly created database root by calling `create_root()` on line 46.
- It associates the root with `rest_dict` by calling `set_value()` on line 47.

Now any applications that need to access `Restaurant` objects can do so, first by retrieving the collection object that is associated with the root and then by using any of the collections methods (for example, `os_Dictionary::pick()`) to access the elements of the collection.

Retrieving elements

The program provides two ways to retrieve elements from the collection:

- By using `os_Dictionary::pick()` to find the name of a restaurant in the dictionary key. This function returns the `Restaurant` object with the matching name, if one exists in the database; see `main.cpp` on page 30, line 39.
- By using an `os_cursor` to locate restaurants with a city member that matches the name of a city; the sample program uses this in the `rlist` application.

The main application (`reserve`) expects the name of a restaurant as one of its command-line arguments. It passes this argument (`argv[1]`) to the `os_Dictionary::pick()` method, which returns a pointer to the `Restaurant` object if an object with a matching `name` member is found. For more information about using cursors and dictionaries, see Chapters 3 and 4 of the *PSE Pro for C++ Collections Guide and Reference*.

Building the Collections Example

Building the collections version of the restaurants reservations application requires the same steps as building the basic PSE Pro version:

1. Run `pssg` to generate the schema.
2. Compile the source files.
3. Link the object files into the executable.

Note that you cannot reuse the schemas that were produced by `pssg` for the noncollections version of the program; see Generating Schemas on page 26. The changes to the `Restaurant` class (for example, the addition of the `city` member) and the addition of an `os_Dictionary` require you to run `pssg` again to generate fresh schemas. The new schema source file looks like this:

schema.scm

```
#include "restaurant.hh"
#include <os_pse/ostore.hh>
OS_MARK_SCHEMA_TYPE(Restaurant);
OS_MARK_DICTIONARY(char*, Restaurant*);
```

Use the `OS_MARK_DICTIONARY()` macro to mark each `os_Dictionary` in the schema source file, specifying the type of the key used by the dictionary and the type of the element stored in the dictionary. For more information about this macro, see `OS_MARK_DICTIONARY()` on page 41.

Command lines

Here are the command lines for generating schemas, compiling, and linking the various applications:

```
pssg -asof schema.obj schema.scm
cl /GX /MD /c init_db.cpp main.cpp restaurant.cpp rlist.cpp add_restaurant.cpp
cl /Fereserve.exe main.obj restaurant.obj schema.obj
cl /Feinit_db.exe init_db.obj restaurant.obj schema.obj
cl /Ferlist.exe rlist.obj restaurant.obj schema.obj
cl /Feadd_restaurant.exe add_restaurant.obj restaurant.obj schema.obj
```

The command lines are similar to those used for building the noncollections version of the applications, the only differences are to build the additional applications `rlist` and `add_restaurant`; see Building PSE Pro Applications on page 26.

Running the application

Assuming that the database contains a `Restaurant` object with its `name` member set to "`Il Falchetto`" and its `tables` member set to at least `5`, here is a sample run:

```
C:\> reserve "Il Falchetto" 17
Reserved 5 tables.
```

An Abbreviated PSE Pro Reference

This section provides reference information for features of the PSE Pro API that are mentioned in this Guide. The information is abbreviated in the number of features described as well as in the level of information for each feature. For example, `os_database::open()` has several overloadings, but only the one used in the example programs presented in this Guide is described here.

In other words, this information is provided solely for the convenience of the reader who wishes to read this Guide without having to look up an API elsewhere in the PSE Pro documentation. However, for detailed and complete information about the PSE Pro API, refer to the *PSE Pro for C++ API User Guide* and to the *PSE Pro for C++ Collections Guide and Reference*.

All reference items are listed alphabetically. Functions are listed by their fully qualified names.

Compiler Options

The following table lists and describes the compiler options used to compile the example programs presented in this Guide

Option	*Meaning*
`/c`	Causes the compiler to compile without linking.
`/GX`	Enables PSE Pro synchronous exception handling.
`/MD`	Defines multithreading macros. You must specify this option even if your application does not use multiple threads.
`/I`	Use this to specify the location of the PSE Pro `include` directory. You can use this option more than once on the same command line to specify additional `include` directories. Necessary if the INCLUDE environment variable is not set to the PSE Pro include directory.

For more information, see the following:

- The *PSE Pro for C++ API User Guide* for detailed information about compiling PSE Pro applications and about other options you might need to use
- Compiling and Linking on page 27 for an example command line

get_os_typespec()

Certain features of the API (for example, persistent `new`) require a typespec argument when you allocate persistent storage for an object. The simplest way to get a typespec is to declare `get_os_typespec()` as a member of your class and implement it using the static function `os_ts<>::get()` as follows (where *`class_name`* specifies the name of the class in which you declared it):

```
static os_typespec* get_os_typespec()
{
  return os_ts<class_name>::get();
```

```
}
```

At run time, you can call `get_os_typespec()` for a pointer to a typespec, as follows:

```
os_typespec* ts = class_name::get_os_typespec();
```

The return value is static and therefore reusable; your application incurs no expense by calling `get_os_typespec()` and does not have to delete the typespec.

The `get_os_typespec()` function is also implemented as a member function of the collection classes (for example, `os_set` or `os_Dictionary`), enabling you to use it to return a typespec argument when you use `::operator new()` to create a collections object.

For more information

See the following:

- `os_typespec::get_char()` on page 42 for information about supplying the typespec argument for fundamental types, such as `char`
- `restaurant.hh` on page 20, line 32, for an example declaration
- `restaurant.cpp` on page 20, line 23, for an example call

objectstore::initialize()

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

This function initializes PSE Pro. It must be called before your application makes any use of PSE Pro functionality.

For more information

See `main.cpp` on page 24, line 12, for an example.

::operator delete()

`::operator delete()` is a global function that you can use to release either transient or persistent storage. PSE Pro determines the type of storage at run time.

The following example releases storage allocated for `object`, regardless of whether the storage is persistent or transient:

```
delete object;
```

The next example releases storage allocated for the array `obj_array`:

```
delete [ ] obj_array;
```

Note

When you use `::operator delete()` to delete a persistent object, you remove the object from persistent storage, in effect making it inaccessible.

For more information

See the destructor `~Restaurant()` in `restaurant.hh` on page 20, line 16, for an example of the `delete` operator used to delete a persistent character array.

::operator new()

PSE Pro overloads the `::operator new()` global function for allocating persistent storage. The syntax is

`new([`*`size`*`, ]`*`placement`*`, `*`typespec`*` [, `*`array-size`*` ] )`

The arguments have the following meanings:

- *`size`* is the `size_t` argument that specifies the size of the object you are creating. This argument is optional and is supplied by the compiler by default.
- *`placement`* is a pointer to a persistent storage unit, in other words, a pointer to an `os_database`.
- *`typespec`* is a pointer to an `os_typespec` object that provides PSE Pro with typing information about the object you want to make persistent.
- *`array-size`* is the number of elements in the array. Specify this argument only if you are allocating persistent storage for an array.

For more information

See the following:

- `os_database::of()` on page 39
- `os_typespec::get_char()` on page 42 for information about typespecs for fundamental types
- `get_os_typespec()` on page 36 for information about typespecs for user-defined types
- `restaurant.cpp` on page 20, line 25, for an example of `new` used to allocate storage for a user-defined type
- `restaurant.cpp` on page 20, line 52, for an example of `new` used to allocate storage for a fundamental type

os_collection::initialize()

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

This function must be called before any use of the collections or relationship facility and after `objectstore:: initialize()` is called.

For more information

See the following:

- `objectstore::initialize()` on page 37 for information about initializing PSE Pro
- `main.cpp` on page 32, line 21, for an example

os_database::close()

```
void close();
```

Closes the database specified by the `this` argument.

For more information

See `main.cpp` on page 24, line 44, for an example.

os_database::create()

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

Returns a pointer to a newly created database, with the specified *pathname*. If *pathname* already exists, an exception is signaled. However, there are overloadings of this function that will cause it to overwrite an existing database at *pathname*.

For more information

See `init_db.cpp` on page 22, line 21, for an example.

os_database::create_root()

```
static os_database_root* create_root(const char* root-name);
```

Returns a pointer to a newly created root in the database specified by the `this` argument. The database root is named *root-name*, enabling you to retrieve it by name. The root and its *root-name* are persistently stored in your database. You use the root to retrieve an associated entry-point object from your database.

For more information

See the following:

- `os_database::find_root()` on page 40
- `os_database_root::set_value()` on page 40 for information about associating the root with an entry-point object
- `restaurant.cpp` on page 20, line 27, for an example

os_database::of()

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

This function can be used as the first argument to `::operator new()` when you want to allocate storage in the database containing the previously allocated *object*. The `of()` function returns a pointer to an `os_database` object.

For more information

See the following:

- `::operator new()` on page 38
- `restaurant.cpp` on page 20, line 52, for an example

os_database::open()

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

Returns a pointer to an existing database, with the specified *pathname*. If *pathname* does not exist, an exception is signaled. However, there are overloadings of this function that will cause it to create a database.

For more information

See `main.cpp` on page 24, line 21, for an example.

os_database::find_root()

```
static os_database_root* find_root(const char* root-name);
```

Returns a pointer to the database root named *root-name* in the database specified by the implicit `this` argument.

For more information

See the following:

- `os_database::create_root()` on page 39 for information about creating a root
- `restaurant.cpp` on page 20, line 39, for an example

os_database_root::get_value()

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

Returns a pointer to an entry-point object associated with the database root specified by the implicit `this` argument. The typespec argument is optional; if specified, PSE Pro checks that it matches the typespec argument specified in `os_database_root::set_value()`. If this argument is unspecified or `0`, PSE Pro does not check for matching typespecs.

For more information

See the following:

- `os_database_root::set_value()` on page 40 for information about associating a root with an entry-point object
- `restaurant.cpp` on page 20, line 41, for an example

os_database_root::set_value()

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

Establishes an association between *object* (a persistent stored object in your database) and the root to which the implicit `this` argument points. The typespec argument is optional; if specified, PSE Pro checks that it matches the typespec argument specified in `os_database_root::get_value()`. If this argument is unspecified or `0`, PSE Pro does not check for matching typespecs.

For more information

See the following:

- `os_database_root::get_value()` on page 40
- `restaurant.cpp` on page 20, line 29, for an example

os_Dictionary::pick()

```
os_Dictionary<class_name>::pick(*key-ptr);
```

Returns a pointer to an element of the implicit `this` argument that has the value of the key pointed to by *key-ptr*. If there is more than one such element, an arbitrary one is picked and returned. If there is no such element, 0 is returned. If the dictionary is empty, returns 0.

For more information

See the following:

- `os_Dictionary` in Chapter 4 of the *PSE Pro for C++ Collections Guide and Reference* for a full description of this function
- `main.cpp` on page 32, line 39, for an example

OS_PSE_END_FAULT_HANDLER

This macro ends the fault handler block established by the `OS_PSE_ESTABLISH_FAULT_HANDLER` macro.

For more information

See `OS_PSE_ESTABLISH_FAULT_HANDLER` on page 41.

OS_PSE_ESTABLISH_FAULT_HANDLER

This macro establishes the fault handler block required by PSE Pro applications so that PSE Pro can detect references to persistent storage. All applications performing PSE Pro operations must use the `OS_PSE_ESTABLISH_FAULT_HANDLER` and `OS_PSE_END_FAULT_HANDLER` macros at the top of every stack in the program. Essentially, this requirement means that the code in the top-level function (`main()` or `WinMain()`) must be contained by these macros. Applications that use multiple threads must also include these macros at the beginning and end of any thread that performs PSE Pro operations.

For more information

See `init_db.cpp` on page 22, line 11 and line 33, for an example.

OS_MARK_DICTIONARY()

This macro is used to mark a dictionary as persistent in schema source files. It has the following syntax:

`OS_MARK_DICTIONARY(*key-type, *class)`

where *`key-type`* is the data type of the key and *`class`* is a pointer to the class (or other element) to be included in the application's schema.

For more information

See the following:

- Chapter 4 of *PSE Pro for C++ Collections Guide and Reference* for more information about the `os_Dictionary` class.
- "`schema.scm`" on page 34 for an example

OS_MARK_SCHEMA_TYPE()

This macro is used to mark a class as persistent in schema source files. It has the following syntax:

`OS_MARK_TYPE(class)`

where *`class`* is the class to be included in the application's schema.

You must use this macro to mark every class on which the application might perform persistent `new`.

For more information

See the following:

- Chapter 2 of *Building Applications with PSE Pro for C++*for more information about classes you must mark
- "`schema.scm`" on page 27 for an example

os_transaction::begin()

Use this method to begin a PSE Pro transaction; use `os_transaction::commit()` to end and commit it. Here is the syntax for defining a transaction:

```
os_transaction* txn = os_transaction::begin("txn-tag", txn-type);
  // your transaction code goes here
os_transaction::commit();
```

The arguments have the following meanings:

- *`txn-tag`* is an identifier that uniquely identifies this transaction. The argument is optional; if you set it, you can retrieve it later with a call to `os_transaction::get_name()`.
- *`txn-type`* indicates whether the transaction performs read/write access to persistent data (`os_transaction::update`), or read-only access (`os_transaction::read_only`).

For more information

See the following:

- Chapter 6 of the *PSE Pro for C++ API User Guide* for detailed information about transactions.
- `main.cpp` on page 24, line 24, for an example of a transaction

os_transaction::commit()

Use this method to end a transaction. Committing a transaction saves all updated information to the database.

For more information

See the following:

- Chapter 6 of the *PSE Pro for C++ API User Guide* for detailed information about transactions.
- `main.cpp` on page 24, line 24, for an example of a transaction

os_typespec::get_char()

```
static os_typespec *get_char();
```

Returns a pointer to a typespec for the type `char`. Typespecs are used by different APIs, including persistent `new`, which requires a typespec as an argument when you allocate persistent storage.

The `get_char()` method is one of a number of `os_typespec` methods that you can use when you are allocating persistent storage for a fundamental type, such as `char`.

For more information

See the following:

- `get_os_typespec()` on page 36 for getting a typespec for a user-defined type
- `::operator new()` on page 38
- `restaurant.cpp` on page 20, line 52, for an example

pssg

The `pssg` utility is used to generate schemas. Here is the format of the `pssg` command line for generating schemas for applications like those presented in this Guide:

`pssg -asof` *`app-schema-obj-file`*`.obj` *`schema-src-file`*`.cpp`

The arguments have the following meanings:

- `-asof` specifies that `pssg` should produce a schema object file named *`app-schema-obj-file`*`.obj`. This object file is ready for linking with your application.
- *`schema-src-file`*`.cpp` specifies the C++ schema source file that you must supply as input to `pssg`. This file does not require an option but must appear last on the command line.

For more information

See the following:

- Generating Schemas on page 26 for information about the schema source file
- Chapter 2 of *Building Applications with PSE Pro for C++* for a full description of `pssg` and all of its options
- "pssg command line" on page 27 for an example `pssg` command line

Index

A

aborting a transaction 12
API
  checklist 12
  overview 10
  reference information 36
application schema object file 27
application schemas 27
applications
  compiling 27
  generating schemas 26
  linking 28
associating roots and objects 40

B

basic operations
  example program 19
  listed 11
`begin()` member of `os_transaction`
  description 42
building applications
  basic PSE Pro 26
  collections 34

C

checklist of PSE Pro operations 12
`close()` member of `os_database`
  description 38
  example 17
`coll.hh` header file 29
`coll\dict_pt.cc` header file 29
`coll\dict_pt.hh` header file 29
collections facility
  example program 29
  overview 28
  requirements 29
`commit()` member of `os_transaction`
  description 42
committing a transaction 12
compiling
  basic application 27
  collections application 35
  `iostreams` application 17
  options 36
`create()` member of `os_database`
  description 39
  example 22
`create_root()` member of `os_database`
  description 39
  example 21

D

database
  basic operations 11
  `os_database` methods 12
database roots
  creating 39
  described 11
  entry-point object 40
  finding 40
  operations 13
  `os_database_root` methods 13
`delete` operator
  description 37
dictionary 34

E

elements of a collection 28
entry-point object
  collection as 33
  described 11
  example 25
  retrieving 40
environment variables

INCLUDE 26
LIB 26
example programs
basic PSE Pro version 19
collections version 29
iostreams version 15

F

find_root() member of os_database
description 40
example 21

G

get_char() member of os_typespec
description 42
example 21
get_os_typespec()
description 36
example 20
get_value() member of os_database_root
description 40
example 21
global functions
operator delete() 37
operator new() 38

H

header files
applications using collections 29
coll.hh 29
coll\dict_pt.cc 29
coll\dict_pt.hh 29
in basic applications 12
in schema source file 26
ostore.hh 12

I

implementing PSE Pro 14
INCLUDE environment variable 26
include files 12
init_db application
basic PSE Pro version 22
iostreams version 17
initialization
collections facility 29
PSE Pro 12
initialize() member of objectstore
description 37
example 22
required 12
initialize() member of os_collection
description 38
example 32
iostreams example 15

L

LIB environment variable 26
linking
application using collections 35
basic application 28

M

macros
OS_MARK_DICTIONARY 41
OS_MARK_SCHEMA_TYPE 41
OS_PSE_END_FAULT_HANDLER 41
OS_PSE_ESTABLISH_FAULT_HANDLER 41
managing database operations 11
marking types 26

N

new operator
description 38
example 21

O

objectstore::initialize()
description 37
example 22
required call 12
of() member of os_database
description 39
example 21
open() member of os_database
description 39
example 24
operator delete()
deleting transient objects 25
description 37
operator new()
description 38
example 21

optimizing
storage allocation 39
options
compiler 36
pssg utility 43
os_collection::initialize()
description 38
example 32
os_database::close()
description 38
example 17
os_database::create()
description 39
example 22
os_database::create_root()
description 39
example 21
os_database::find_root()
description 40
example 21
os_database::of()
description 39
example 21
os_database::open()
description 39
example 24
os_database_root::get_value()
description 40
example 21
os_database_root::set_value()
description 40
example 21
os_Dictionary::pick()
description 40
OS_MARK_DICTIONARY macro
description 41
example 34
OS_MARK_SCHEMA_TYPE macro
description 41
example 27
OS_PSE_END_FAULT_HANDLER macro
description 41
example 22
OS_PSE_ESTABLISH_FAULT_HANDLER macro
description 41
example 22
os_transaction::begin()
description 42
example 22
os_transaction::commit()
description 42
example 22
os_typespec::get_char()
description 42
example 21
ossg utility
example command line 27
ostore.hh header file
in basic applications 12
in collections applications 29
in schema source file 26

P

page-fault macros 12
OS_PSE_END_FAULT_HANDLER 41
OS_PSE_ESTABLISH_FAULT_HANDLER 41
persistence
creating persistent objects 11
operator delete() 37
operator new() 38
overview 10
page-fault macros 12
pick() member of os_Dictionary
description 40
populating a collection 28
PSE Pro
API 10
basic operations 11
building an application 26
pssg utility
description 43

R

reservations application
basic PSE Pro version 19
collections version 29
iostreams version 15
rolling back a transaction 12

S

schema source file
checking for errors 27
example 27
OS_MARK_DICTIONARY 41
OS_MARK_SCHEMA_TYPE macro 41

schemas 26
set_value() member of os_database_root
  description 40
  example 21

T

transactions
  aborting 12
  committing 12
  delimiting 13
  example 22
  managing 12
  rolling back 12
transient objects
  compared to persistent objects 10
  deleting 25
typespec argument
  new 38
  os_database_root::get_value() 40
  os_database_root::set_value() 40
typespecs
  fundamental types 42
  get_os_typespec() 36
  os_typespec::get_char() 42
  reasons for using 23
  user-defined types 36

U

utilities
  pssg 43